# DECENTRALIZED COMPUTATION OF QUALITATIVE SPATIAL RELATIONSHIPS IN MOBILE GEOSENSOR NETWORKS

**ALAN JAMES BOTH**
0000-0003-4021-9632

Submitted in total fulfillment of the requirements of
the degree of Doctor of Philosophy

September 2015

Department of Infrastructure Engineering
The University of Melbourne

Produced on archival quality paper

## ABSTRACT

Consider the task of monitoring changes in the structure of dynamic spatial phenomena such as algal blooms or oil spills. Such phenomena consist of multiple disconnected region components, which can reconfigure over time. This work uses qualitative spatial reasoning to record only salient changes to the internal structure of these regions.

To detect and store such qualitative spatial information, this research proposes a collection of five in-network, decentralized algorithms. Unlike previous work, these algorithms are able to operate in networks of mobile geosensor nodes with no access to coordinate information. A decentralized approach to algorithm design allows for information processing to take place within the network, with the defining feature being that no single node has access to the entirety of data in the network.

A fundamental aspect of decentralized algorithms is wireless communication between nodes. As such communication has high power requirements, the algorithms presented in this work are designed to minimize the amount of communication taking place, while still maintaining accuracy. Experimental evaluation of these algorithms found that meeting the criteria of sufficient node density, broadcast interval, and communication distance produced accurate results. These three factors were found to be dependent upon the characteristics of the phenomena being monitored.

It was also found that the modules comprising each of the decentralized algorithms exhibited either sub-polynomial, linear, or weakly polynomial scalability (with the worst case being $O(n^{1.1})$). The order of scalability produced by a module was found to be due to the type of decentralized algorithm that module was based on, with leader election based algorithms producing weakly polynomial scalability, and surprise flooding based algorithms producing linear or sub-polynomial scalability.

This work aims to provide long-term environmental monitoring to areas that have previously been unable to be monitored due to their location, the cost of deploying a suitable geosensor network, or the time-span required.

# DECLARATION

The following declaration page, signed by the candidate:

This is to certify that:

1. the thesis comprises only my original work towards the PhD except where indicated in the Preface,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is fewer than 100 000 words in length, exclusive of tables, maps, bibliographies, and appendices.

*Melbourne, September 2015*

Alan James Both

# PREFACE

This thesis is supported by funding from the Australian Research Council (ARC) under the Discovery Projects Scheme, grant number DP120103758. Additionally, this thesis is based on published works from my PhD research during candidature. As such, some ideas, figures, and algorithms have appeared previously in the following publications:

[1] A. Both and M. Duckham, "Qualitative Spatial Structure in Complex Areal Objects Using Location-Free, Mobile Geosensor Networks," in *2013 IEEE 13th International Conference on Data Mining Workshops (ICDMW)*, Dec. 2013, pp. 978–985.

[2] A. Both, W. Kuhn, and M. Duckham, "Spatiotemporal Braitenberg Vehicles," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. SIGSPATIAL'13. New York, NY, USA: ACM, Nov. 2013, pp. 74–83. [Online]. Available: http://doi.acm.org/10.1145/2525314.2525344

[3] A. Both, M. Duckham, P. Laube, T. Wark, and J. Yeoman, "Decentralized Monitoring of Moving Objects in a Transportation Network Augmented with Checkpoints," *The Computer Journal*, vol. 56, no. 12, pp. 1432–1449, Dec. 2013. [Online]. Available: http://comjnl.oxfordjournals.org/content/56/12/1432

## OTHER PUBLICATIONS

[4] A. Galton, M. Duckham, and A. Both, "Extracting Causal Rules from Spatio-temporal Data," in *Spatial Information Theory: 12th International Conference, COSIT*, Santa Fe, NM, USA, Oct. 2015.

[5] S. Li, Z. Long, W. Liu, M. Duckham, and A. Both, "On redundant topological constraints," *Artificial Intelligence*, vol. 225, pp. 51–76, Aug. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370215000569

[6] M. Duckham, A. Galton, and A. Both, "Identifying perpetuation in processes driving fish movement," in *In 23rd GIS Research UK (GISRUK) conference*, Leeds, UK, Apr. 2015. [Online]. Available: http://leeds.gisruk.org/abstracts/GISRUK2015_submission_87.pdf

# ACKNOWLEDGMENTS

I would first like to express my deepest gratitude to my supervisor, Prof. Matt Duckham. His continued support, guidance and enthusiasm provided me with abundant motivation and inspiration. Words cannot adequately express what a privilege it has been to work with him.

I would also like to thank my thesis advisory committee; Assoc. Prof. Allison Kealy and Prof. Stephan Winter. Their kind words and unique insights encouraged me consider my work from all angles, and my work has improved considerably for it.

To my colleagues in the Ambient Spatial Intelligence (AmSi) group; Dr. Susanne Bleisch, Dr. Lin-Jie Guan, Dr. Myeonghun Jeong, Mellissa Shahrom, Jeremy Yeoman, Azadeh Mousavi, Lisa Cheong, and Farzad Almadara; you have my sincere thanks. I've greatly enjoyed our time spent together, and our discussions have broadened my knowledge in so many ways. Having the opportunity to present my work to such kind people has made presenting at conferences a much easier task. In particular, I would like to thank Prof. Susanne Bleisch. Our discussions on data visualization have sparked a keen interest in me. Her shrewd advice has greatly improved both my knowledge and skills in this area.

To my teaching colleagues at the University of Melbourne; Dr. Graham Brodie, Victoria Petrevski, and Isabel Pacheco; teaching with you has been a wonderful and supremely rewarding experience. I've never felt more exhausted and energized than in the classroom. You have instilled a love of teaching in me that I won't soon forget.

To my friends, thank you for your patience and support. Without your continued efforts to socialize with me, I would now most likely be a hermit.

Lastly, I would like to thank my parents Eddie and Linda, and my sister Amanda. Their love and support has always been a great joy to me.

# CONTENTS

# LIST OF FIGURES

## LIST OF TABLES

## LIST OF ALGORITHMS

# 1 INTRODUCTION

The range of use cases of geosensor networks has expanded widely in recent years, particularly with the advent of decentralized algorithms, which allow the processing and analysis of environmental data to take place within the network. Presently, many of the decentralized algorithms at the forefront of the field have requirements that severely limit their usefulness. Two restrictions that pervade the majority of these algorithms are as follows:

1. The nodes comprising the network must know their precise coordinates, and,
2. The nodes must remain static throughout the algorithm's operation.

This thesis focuses on eliminating these restrictions, by developing efficient decentralized algorithms for monitoring the qualitative spatial structure of regions using location-free geosensor networks with nodes that are mobile.

## 1.1 BACKGROUND AND PROBLEM STATEMENT

To illustrate the need for decentralized algorithms that make use of node mobility and operate without the coordinates of the nodes, the motivating example of monitoring the internal structure of regions will be used.

Monitoring areas of interest is the primary application of geosensor networks. Depending on the specific use, areas of interest can include regions of high temperature, flooding, chemical or biological contaminants. Each node in the geosensor network is outfitted with a sensor or combination of sensors capable of detecting these areas of interest. This is typically done by setting a threshold; for example, a region of high temperature could be defined by the nodes as anything above 30 degrees. The task of monitoring these regions can then be simplified by treating them as single homogeneous areas that require both the boundaries and interiors of the regions to be contiguous (i.e., no disconnected regions or holes).

This simplicity however does not take into account the inherently complex internal structure of many phenomena worth monitoring, both natural and artificial in origin. For example: algal blooms, regions of abnormal temperature, oil spills, and bushfires can all contain disconnected parts, holes and islands. Regions such as these,

with their topologically complex interiors, are referred to as complex areal objects. Additionally, geosensor networks are expected to monitor regions over time. For simple regions, this involves tracking changes in location and shape of the region. For complex areal objects, changes to the internal structure must also be monitored.

While the monitoring of areas could be carried out using quantitative spatial representations such as raster grids, this project will instead focus on a more abstract, symbolic representation. Specifically, the work will rely on qualitative spatial reasoning (QSR), which allows the structure of regions to be modeled algebraically as a series of relations [13].

As the size of a geosensor network deployment increases, so too does the frequency of node failure. Because of this, algorithms must be designed to take into account the addition and removal of nodes throughout their operation. In addition to dynamic regions, the nodes themselves may be mobile. For example, when monitoring algal blooms, nodes may be stationed on untethered buoys, which can move due to ocean currents and wind effects [14]. In fact, such types of effects may even be used for network distribution.

The algorithms used to carry out these tasks are decentralized in nature. The key difference between decentralized and centralized approaches on geosensor networks lies in where the data obtained from the sensors is processed and stored. A centralized approach requires that all sensor data be transferred to and aggregated at a central location before processing can take place. A decentralized approach however allows for processing to take place within the network, with the essential feature being that no single node has access to the entirety of data in the network.

The decentralized approach requires that each node within the network has a CPU for processing data, a wireless radio for communicating data, and sensors for generating data. Given that nodes on geosensor networks have limited battery life, choices must be made in order to prolong the operating lifespan of a network. This is done by carefully selecting the type and amount of sensors used in each node as well as the type of algorithms used.

One class of sensors with high power requirements are location sensors. These are typically a type of radio that receives information from transmitters with known locations in order to compute the node's coordinates. The most widely used example of this would be Global Navigation Satellite Systems (GNSS) such as GPS. Because of this, algorithms will be designed for nodes that are not equipped with location sensors. That is not to say that the nodes will be location-free in that they will not make use of location information, just that they will not have access to their absolute coordinates. Nodes will still be able to make use of relative location by being given a uniform communication range.

A uniform communication range means that for any given node there is a set of neighboring nodes within this range that the node can communicate with directly, and a set of nodes which are further away that require messages to be passed through intermediary nodes. In fact, the number of nodes a message has passed through can be used as a metric for the relative distance that message has traveled.

Additionally, nodes will be equipped with a single threshold-based sensor capable of detecting only the presence or absence of the region. This will further reduce the power requirements of the nodes as well as the complexity of the algorithms. Wireless communication also requires large amounts of power but is a fundamental requirement for decentralized algorithms. With this in mind, algorithms must be designed in such a way as to minimize the amount of communication taking place.

## 1.2 HYPOTHESIS AND SCOPE

It is the focus of this thesis that qualitative spatial information is extracted from regions that are monitored by dynamic geosensor networks. Given these specifications and constraints, this research will address the following hypothesis:

HYPOTHESIS That decentralized algorithms can be designed to detect and monitor a variety of complex spatial phenomena running on geosensor networks that:

1. Operate without coordinate information,
2. Are tolerant of geosensor node mobility,
3. Accurately detect the qualitative structure of the underlying region,
4. Correctly detect salient changes to the qualitative structure of dynamic regions, and
5. Are efficient in the amount of communication needed to perform their task.

In order to test this hypothesis, the following methodology will be used: Firstly, a formal model must be selected for both the decentralized algorithms and the geosensor network they run on. Next, these decentralized algorithms will be implemented in an agent-based simulation system before being evaluated based on the criteria of scalability and veracity.

To define the scope of this study, this research proposes four classes of decentralized algorithms based on whether the region being monitored is static or dynamic and whether the region's structure is simple (i.e., the region contains a single hole) or complex (i.e., the region may contain multiple holes).

## 1.3 SIGNIFICANCE OF STUDY

As mentioned in the previous section, the primary outcome of this thesis is the specification and evaluation of decentralized algorithms that describe the qualitative structure of regions using location-free geosensor networks where the nodes are mobile. The key results of this research are:

- A collection of decentralized algorithms that efficiently record the internal structure of a monitored static or dynamic region with a simple or complex internal structure.
- A formal structure capable of efficiently storing the qualitative internal structure of, and changes to these regions.
- A set of qualitative spatial relations to distinguish when a region component is completely or partially enclosed by another region component, or partially enclosed by multiple region components.
- A conceptual neighborhood graph for dynamic regions to describe the specific way region configurations enter and exit the described qualitative spatial relations in order to provide a detailed description of how the relations between region components change over time.

This research aims to contribute to the field of environmental monitoring by bringing long-term monitoring to areas previously unsuited to geosensor networks and at a reduced cost. These nodes would require only a single sensor to detect the presence or absence of the monitored phenomena and a wireless transceiver to communicate with their neighbors. This, in addition to not including a GNSS radio, would reduce both the cost and the power requirements of the network.

## 1.4 STRUCTURE OF THESIS

The remainder of this thesis is structured as follows: Chapter 2 begins with an overview of qualitative spatial reasoning with respect to regions, specifically covering topological relations between and topological changes to regions. A detailed examination of the maptree, a formal structure capable of efficiently storing the qualitative internal structure of regions, is then discussed. Finally, a selection of decentralized algorithms are presented and categorized based on the specific aspect of the hypothesis they are capable of addressing. The chapter then concludes by identifying the gap in current knowledge this research aims to fill.

Chapter 3 describes the research framework used in this work; namely the formal specification of the geosensor network in addition to the specification, implementation, and evaluation of decentralized algorithms. Chapter 4 presents three decentralized algorithms that are capable of determining the internal structure of static regions and discovering any qualitative relations that may be present. Chapter 5 extends these algorithms, with two decentralized algorithms that are capable of additionally functioning on dynamic regions.

Chapter 6 presents experimental evaluations of the algorithms presented in Chapters 4 and 5, with their performance evaluated in terms of veracity and scalability. Lastly, Chapter 7 presents conclusions based on the results obtained from Chapter 6.

# 2 LITERATURE REVIEW

The purpose of this literature review is to illustrate the components needed to construct algorithms capable of testing the hypothesis. The review begins with an overview of qualitative spatial reasoning with respect to regions, specifically covering topological relations between and topological changes to regions. A detailed examination of the maptree, a formal structure capable of efficiently storing the qualitative internal structure of regions, is then discussed. Finally, a selection of decentralized algorithms are presented and categorized based on what particular aspect of the hypothesis they are capable of meeting.

## 2.1 QUALITATIVE REPRESENTATION OF REGIONS

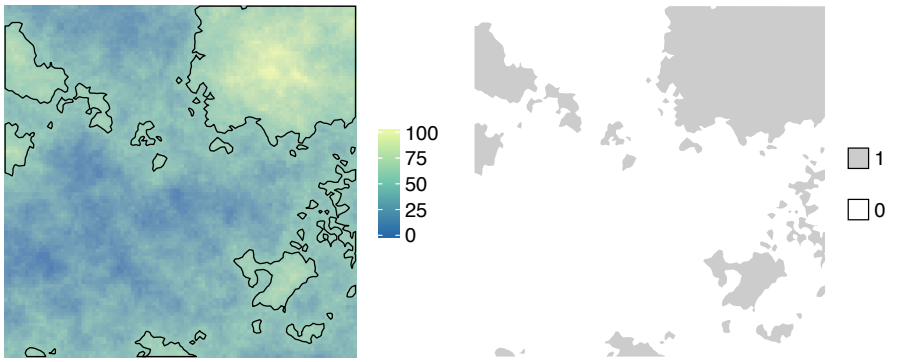
Before discussing decentralized algorithms and the data models they are built upon, it is necessary to first consider the way in which the internal structure of a monitored region is described. The underlying phenomena can be considered a continuous field in geographic space, and samples of this field can be recorded as a scalar quantity by a geosensor network using sensors equipped to nodes within the network. Specifically, this field is recorded using the function $sense : V \to C$ where domain $V$ is the set of nodes in the network and $C$ is the codomain recording their sensed attribute.

### 2.1.1 *Simplification of sensor data*

While some sensors, such as those used to detect algal blooms, may express their results as a Boolean where the sensor either detects or does not detect the presence of an algal bloom (i.e., $sense : V \to \{0,1\}$); many other sensors, such as temperature and pressure sensors, express their results as a scalar quantity (i.e, $sense : V \to \mathbb{R}$). In order develop a qualitative spatial description of the field, the scalar field recorded by the node's sensors must be simplified. This research will do so by modifying the sensor so that it records a Boolean value (i.e., $sense : V \to \mathbb{R} \to \{0,1\}$), splitting the field into a collection of high or low intensity regions based on a threshold value.

An example of thresholding continuous fields is shown in Figure 1, where the threshold value has been set to 50, with areas above this value considered to be within the region and areas below this value outside the region. Adopting a threshold value is however not the only option for categorizing continuous fields. For example, the egg-yolk model [15] adopts a third intermediate case for areas that may

or may not be within the region. For the sake of simplicity, this work will focus on the crisp boundaries induced by threshold values.




a. Continuous field with threshold boundaries

b. Resulting regions

Figure 1: Randomly generated continuous field split into regions with values greater than 50.

### 2.1.2 *Topological relations between regions*

Now that a mechanism has been established for the detection of regions by a geosensor network, relations between these regions can be considered. Specifically, this work makes use of qualitative spatial reasoning to model the topological relations between regions as a series of algebraic relations [13].

There are two methods to consider when modeling topological relations between regions; an axiomatic approach and a point-set approach. Both methods are capable of representing the same topologically distinct relations between two regions, as illustrated by Figure 2. The axiomatic approach is best described by region connection calculus (RCC) [16–18], where all relations are constructed from the binary relation *connectedness*.

Specifically, consider the connected topological space $X$ where $U$ is the set of nonempty regular closed sets of $X$. Each element in $U$ is a region, which may have holes or multiple connected components. For any two regions a and b, a is *connected to* b (i.e., a**C**b) if $a \cap b \neq \varnothing$. Building on this relation, region a is said to be *part of* b (i.e., a**P**b) if $a \subseteq b$. From these two relations, the set of eight relations known as RCC8 can be constructed, as shown in Figure 2. These relations are: x**DC**y, where x is disconnected from y; x**EC**y, where x is externally connected to y; x**PO**y, where x partially overlaps y; x**EQ**y, where x and y are identical; x**TPP**y, x is a tangential proper part of y; x**TPPI**y, where x is the inverse tangential part of y; x**NTPP**y, where x is a non-tangential proper part of y; and x**NTPPI**y, where x is the inverse non-tangential part of y.

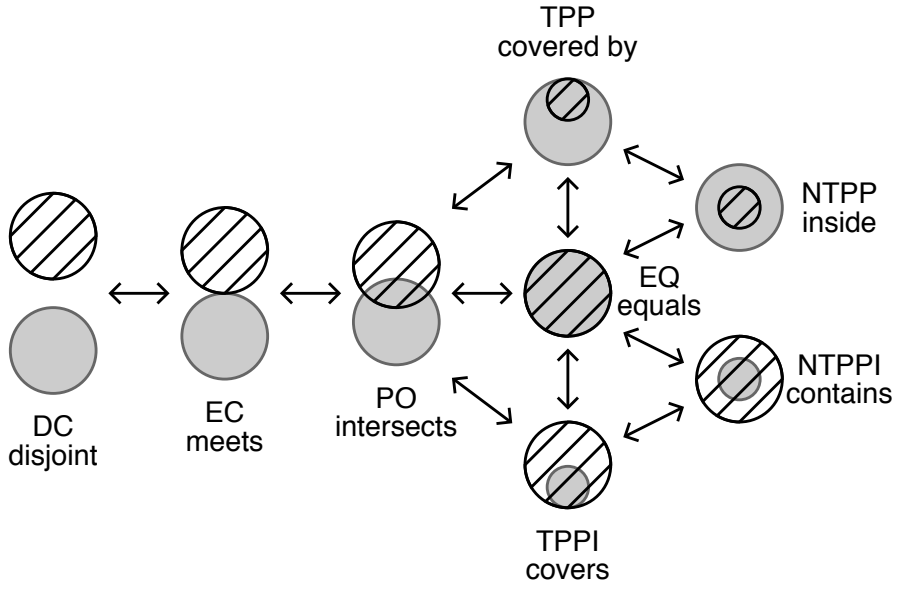



Figure 2: Eight distinct topological relations between black striped region x and shaded grey region y. RCC8 and 4IM/9IM labels are provided. Arrows indicate valid topological transitions between relations.

The axiomatic approach of the region connection calculus uses *connectedness* for its fundamental primitive, considering both simple and complex regions as a whole. Applying this model to a geosensor network would require computing which subsets of nodes are path connected (i.e., there exists a chain of connected nodes between them); an approach that is well suited to this work.

The alternative method of using point-sets is best described by the 4-intersection model (4IM) [19]. This model uses the intersections between point sets as its fundamental primitive; specifically, the intersections between the point sets of the regions' interiors ($^\circ$) and boundaries ($\partial$). The relations between any two regions a and b are then described using the 4-tuple ($\partial a \cap \partial b$, $\partial a \cap b^\circ$, $a^\circ \cap \partial b$, $a^\circ \cap b^\circ$). This model was later extended to the 9-intersection model (9IM), additionally including the point sets of regions' complements ($^-$) to form a 9-tuple [20].

In either intersection model, eight valid relations are provided when considering the topological relations between two regions, as shown in Figure 2. It is important to note that unlike the region connection calculus, the intersection models are only designed for simple regions (i.e., homeomorphic to a disk, containing no holes or disconnected components). While some extensions to the intersection models have been proposed [21, 22] to distinguish topological relations between complex regions, the axiomatic approach of the region connection calculus is better suited for this work.

### 2.1.3 *Topological change*

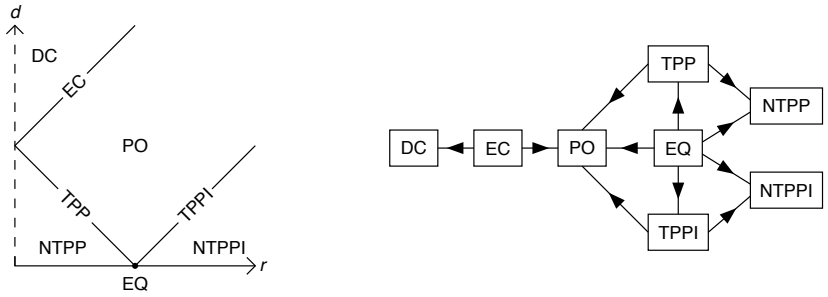
Consider the eight distinct topological relations between a pair of regions presented in the previous section. Assuming these regions were capable of movement, there must be a way for the regions to transition between any pair of topological relations. Looking at Figure 2, it is clear that some relations have more in common than others. For example, the RCC8 relations TPP and NTPP have a greater similarity than the relations DC and EQ. Freska, in his 1992 paper "Temporal reasoning based on semi-intervals" [23] introduces the notion of a conceptual neighborhood, where "two relations between pairs of events are (conceptual) neighbors, if they can be directly transformed into one another by continuously deforming (i.e. shortening, lengthening, moving) the events (in a topological sense)." These conceptual neighbors are illustrated in Figure 2 with black arrows.

A conceptual neighborhood graph of topological relations was first conceived of by Randell and Cohn [16] and later formalized with respect to the RCC8 relations by Randell et al. [17], where it indicated that, like the relations themselves, the direct topological transitions can also be derived axiomatically.

The conceptual neighborhood graph of the RCC8 relations was introduced along with these relations by Randell et al. [17], where it indicated that, like the relations themselves, the direct topological transitions can also be derived axiomatically. Galton [24] first derives the transitions by placing the RCC8 relations within a dominance space by knowing the amount of A inside B and B inside A (discretized to all, some, or none), and whether A and B share any boundary points. This dominance space then draws possible transitions between relations using dominance relations. To clarify dominance relations, consider the transition between DC and PO relations. At some point, the boundaries of the regions must intersect (i.e., the EC relation) before their interiors can. At this instant, neither the DC nor PO relations hold true, meaning that the EC relation dominates both the DC and PO relations, as shown in Figure 3.b.

Galton also describes the conceptual neighborhood by partitioning the RCC8 relations into a phase diagram [7]. This diagram uses two circular regions, one with a fixed radius and position, and the other with a variable radius and position. Plotting the distance between the two regions against the second region's radius produces the phase diagram shown in Figure 3.a. Applying this partition information produces the mode space diagram as previously shown in Figure 3.b.

Egenhofer and Al-Taha, in their 1992 paper "Reasoning about Gradual Changes of Topological Relationships" [25], construct the conceptual neighborhood graph of the 9IM using closest topological distance as a metric. Topological distance was calculated by finding the symmetric difference between two topological relations' 9-tuple




a. Phase diagram partitioning continuous region-region relations into discrete RCC8 relations.

b. Resulting mode space showing dominance relations.

Figure 3: Conceptual neighborhood of RCC8 relations, adapted from [7].

values. For example, the difference between inside (010010111) and contains (001111001) is 6, as digits 2–4 and 6–8 are different. To construct the conceptual neighborhood graph, edges are added for each relation, connecting it to the relation with the smallest topological distance. This adds all edges present in Figure 2 bar the edges between the equal relation and overlaps, inside, and contains. These three edges are then added to the graph as their topological distance is no greater than the shortest path present (e.g. the distance between overlaps and equal is 6, which is also the total distance of traversing overlaps→covers→equals).

Recall from section 2.1.1 that thresholding partitions the space into a collection of components that are either inside or outside the region. In a complex areal object, these are known as the positive and negative region components respectively. Jiang and Worboys [26] categorize basic topological change according to the alterations possible to a complex areal object's containment tree. These five categories are: *insert*, where a region appears; *merge*, where regions combine; *delete*, where a region disappears; *split*, where a region separates into distinct components; and *no change*, where the region remains unaltered. Expanding on these categories, they define six fundamental topological events to cover complex topological changes. The first four are the basic topological events *appearance*, *disappearance*, *merge* and *split*. The last two are *self-merge* and *self-split*, which occur when a region merges or splits with itself, causing an adjacent region to split or adjacent regions to merge.

While it is also possible to define topological events by changes that occur along the edge of a region, this work will restrict topological events to those that occur at a single point, as this method is better suited to the properties of geosensor networks. Figure 4 provides examples of these topological events.

For appearance and disappearance events, this would occur when a single node detects a new region component and when the last node

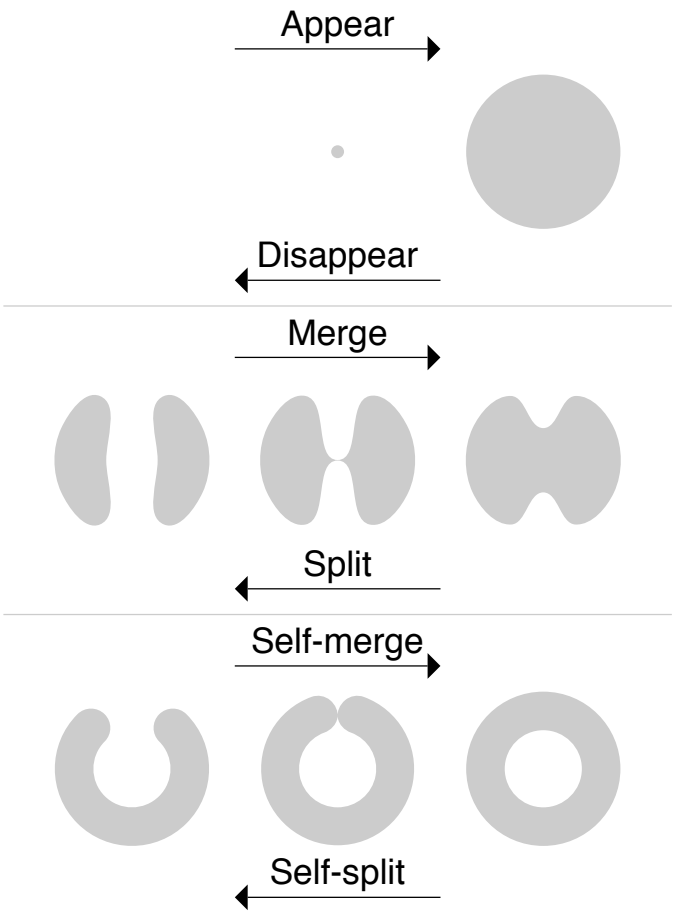



Figure 4: Types of topological events possible for dynamic regions where the event is instigated at a single point.

detecting a region no longer detects that region. For merge events this would mean that at some point a node has detected a connection between the two regions, and for split events, this would occur when the last node connecting the region no longer detects that connection, splitting the region into multiple separate components. Self-merge and self-split events do not need to be explicitly detected as they coincide with split and merge events of adjacent regions.

## 2.2 MAPTREE

The maptree is a formal structure that is that is well suited to describing topological change. This model was introduced by Worboys in his 2011 paper "Modeling indoor space" [27] and further detailed in his 2012 paper "The maptree: A fine-grained formal representation of space" [28]. Specifically, the maptree is a black-white edge labeled tree based on combinatorial maps and adjacency trees capable of uniquely representing the topological structure of regions.

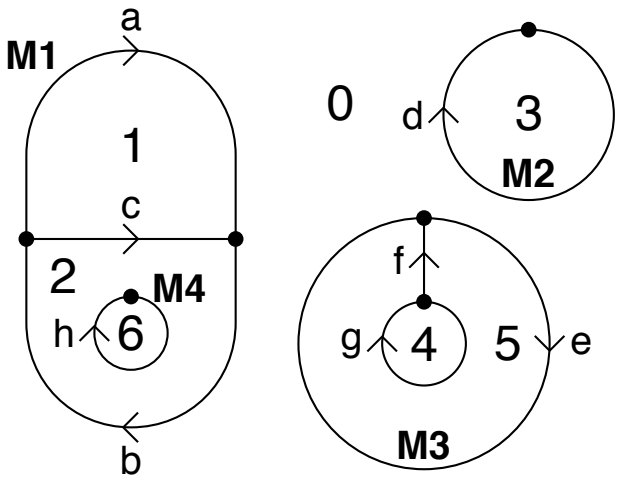



Figure 5: Example region. Black lines represent boundaries between region components.

Consider the region configuration shown in Figure 5, where the region has been subdivided into a collection of region components (or faces) numbered 1–6, by a series of nodes and directed edges labeled a–h. Given that this region configuration is composed of a set of disconnected nodes and edges, it can also be considered a planar embedding of a disconnected graph.

One of the components used to construct a maptree is a combinatorial map, which provides a unique symbolic representation of a connected graph. A combinatorial map is constructed from a set of halfedges ($S$), permutations or cycles of these halfedges ($\alpha$), and involutions or twins of these halfedges ($\tau$). Together these produce the triple $\mathrm{M}\langle S, \alpha, \tau\rangle$. As the the region configuration of Figure 5 consists of multiple connected components, each connected component will be given its own combinatorial map (i.e., M1–M4).

For example, the combinatorial map of the first connected component, M1, the halfedges present are: $S = \{\mathrm{a}, \overline{\mathrm{a}}, \mathrm{b}, \overline{\mathrm{b}}, \mathrm{c}, \overline{\mathrm{c}}\}$. As an edge can bound two faces, each edge has been stored as two halfedges facing opposite directions. For clarity, only one half of each halfedge is labeled in Figure 5, with the arrow indicating its direction.

The cycles of M1 are as follows: $\alpha = (\mathrm{a}, \mathrm{b})(\overline{\mathrm{a}}, \mathrm{c})(\overline{\mathrm{b}}, \overline{\mathrm{c}})$. Each cycle defines the ordering of halfedges around a particular face, where the face is the region component to the left of each halfedge. For example, the cycle (a, b) describes the path of halfedges needed to describe face 0 whereas cycles $(\overline{\mathrm{a}}, \mathrm{c})$ and $(\overline{\mathrm{b}}, \overline{\mathrm{c}})$ describe faces 1 and 2 respectively.

The final component of the combinatorial map is the involution or twin of the halfedges, which are as follows: $\tau = (\mathrm{a}, \overline{\mathrm{a}})(\mathrm{b}, \overline{\mathrm{b}})(\mathrm{c}, \overline{\mathrm{c}})$. For clarity, overline notation has been used to indicate the involution of a halfedge, i.e., $\overline{\mathrm{a}}$ is the twin of a, which borders region components 1 and 0 respectively.

Calculating the combinatorial maps for each of the connected components would produce:

**M1** $\langle\{a, \bar{a}, b, \bar{b}, c, \bar{c}\}, (a, b)(\bar{a}, c)(\bar{b}, \bar{c}), (a, \bar{a})(b, \bar{b})(c, \bar{c})\rangle$,

**M2** $\langle\{d, \bar{d}\}, (d)(\bar{d}), (d, \bar{d})\rangle$,

**M3** $\langle\{e, \bar{e}, f, \bar{f}, g, \bar{g}\}, (e)(\bar{a}, c)(\bar{e}, \bar{f}, g, f), (e, \bar{e})(f, \bar{f})(g, \bar{g})\rangle$,

**M4** $\langle\{h, \bar{h}\}, (h)(\bar{h}), (h, \bar{h})\rangle$.

The second component used to construct a maptree is the adjacency tree, which is a rooted black-white tree that makes use of the adjacency between region components, enabling them to be nested. An alternative method for representing containment relations between region components is to use a containment tree [29]. It is important to note that for binary images, adjacency strictly implies containment [30, 31]. This means that for regions with only positive and negative region components, the containment trees and adjacency trees are identical. Later sections of this work will exclusively use the term containment tree to better distinguish between adjacency relations and containment relations.

In the case of a maptree, the black nodes represent connected components whereas the white nodes represent region components, meaning that edges represent instances where a region component borders a connected component. The root of the tree is a white node that represents the exterior of the region, which in this case is region component 0. Figure 6 shows the completed maptree where it can be seen from its structure that there are edges between region components 0 and 1 and connected component M1, indicating that these two region components are adjacent. As there are no connected components between region components 0 and 6, these two region components are not adjacent.

At this point, the maptree represents the relations between region components and connected components, but does not yet represent the topology of the connected components. To account for this, the cycles of the connected components provided by the combinatorial maps will be used. Recall that the cycles define the ordering of halfedges around a particular region component. As such, each edge will be labeled by the cycle of the connected component that describes that region component. For example, cycle (a, b) of connected component M1 describes the boundary between M1 and region component 0, and so will be placed on edge (0, M1).

### 2.2.1 *Maptree storage*

For the maptree to be of use in this work, it must be possible to store its information compactly within a data structure so that the nodes

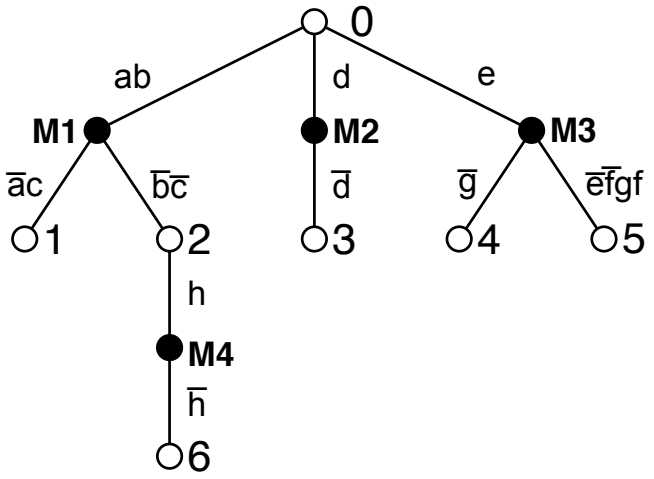



Figure 6: Maptree of the region configuration of Figure 5.

within the geosensor network can make use of it. The DCEL (Doubly-Connected Edge List) [32, 33] is a widely used data structure for the storage of planar embeddings of graphs. Recall that the region configuration shown in Figure 5 is a planar embedding of a disconnected graph, meaning that if it is possible to store this information in a DCEL table, it is also possible to generate a maptree from the DCEL table.

To explain the components of DCEL, table 1, which is based on the region configuration of Figure 5, will be used as an example. A standard DCEL table uses the following columns: halfedge, twin, next, previous and face. Each edge has been stored as two halfedges, with the halfedge's id stored in the halfedge column and its involution's id stored in the twin column. The face column then stores the id of the face or the region component to the left of the halfedge. For example, region component 0 is to the left of halfedge a, whereas region component 1 is to the left of halfedge $\bar{a}$.

The next and previous columns store the ids of the halfedges that immediately succeed and precede the current halfedge. This is done by performing an anticlockwise traversal of the halfedge's face. For example, given that halfedge f is adjacent to face 5, the next halfedge is $\bar{e}$ whereas the previous halfedge is g. With these five columns, it is possible to completely represent the region configuration shown in Figure 5.

Now that the DCEL has been described, it can be used to construct the maptree. The steps necessary to generate a maptree are described in algorithm 1. The *generateMaptree* function of this algorithm can be broken into two stages, with the first assigning a connected component number to each record's component column in the DCEL table (lines 2–7) and the second constructing the maptree as a graph (lines 8–11).

| Halfedge | Twin | Next | Previous | Face | Component |
|---|---|---|---|---|---|
| a | $\bar{a}$ | b | b | 0 | M1 |
| $\bar{a}$ | a | c | c | 1 | M1 |
| b | $\bar{b}$ | a | a | 0 | M1 |
| $\bar{b}$ | b | $\bar{c}$ | $\bar{c}$ | 2 | M1 |
| c | $\bar{c}$ | $\bar{a}$ | $\bar{a}$ | 1 | M1 |
| $\bar{c}$ | c | $\bar{b}$ | $\bar{b}$ | 2 | M1 |
| d | $\bar{d}$ | d | d | 0 | M2 |
| $\bar{d}$ | d | $\bar{d}$ | $\bar{d}$ | 3 | M2 |
| e | $\bar{e}$ | e | e | 0 | M3 |
| $\bar{e}$ | e | $\bar{f}$ | f | 5 | M3 |
| f | $\bar{f}$ | $\bar{e}$ | g | 5 | M3 |
| $\bar{f}$ | f | g | $\bar{e}$ | 5 | M3 |
| g | $\bar{g}$ | f | $\bar{f}$ | 5 | M3 |
| $\bar{g}$ | g | $\bar{g}$ | $\bar{g}$ | 4 | M3 |
| h | $\bar{h}$ | h | h | 2 | M4 |
| $\bar{h}$ | h | $\bar{h}$ | $\bar{h}$ | 6 | M4 |

Table 1: DCEL table augmented with connected component labeling. Table shows entries for Figure 5.

For its input, this algorithm takes a standard DCEL table with an additional component column with records that have been initialized to $\varnothing$. After processing these records, the algorithm outputs a standard maptree in the form of a graph $G = (V, E)$ where $V$ represents the set of black and white nodes comprising the maptree and $E$ represents the set of edges (i.e., $E \subseteq V \times V$). Additionally, these edges are equipped with a *label* attribute, which is used to store the cycle of halfedges from the edge's connected component that describes the edge's region component.

The first stage of the *generateMaptree* function begins by setting the connected component iterator $n$ to 0, which will be incremented every time a new connected component is discovered (line 2). While each connected component will be assigned a number, it is important to note that, like table 1, the M prefix will be added so that the connected components are not confused with the region components.

Each record of the DCEL table is then iterated through (line 3). If a record is found with no entry in its component column, $n$ is incremented and set as the component for that record (lines 4–6). Using record a as an example, the component column would be set to M1.

The *assignComp* function is then applied to that record (line 7). This function is recursive, with the goal of seeking out all records within the same connected component. First, all records within the nomi-

**Algorithm 1** Maptree generation algorithm

```
1:  generateMaptree
2:      let n := 0
3:      for all i ∈ DCEL do
4:          if i.component = ∅ then
5:              set n := n + 1
6:              set i.component := Mn
7:              assignComp(i)
8:      for all j ∈ DCEL do
9:          if edge(j.face, j.component) ∉ G then
10:             insert edge(j.face, j.component) INTO G
11:             set edge(j.face, j.component)label := cycle(j).halfedge
12: twin(record)
13:     return SELECT * FROM DCEL WHERE twin = edge.halfedge
14: next(record)
15:     return SELECT * FROM DCEL WHERE next = edge.halfedge
16: cycle(startRecord)
17:     let currentRecord := startRecord
18:     let items := {startRecord}
19:     while next(currentRecord) ≠ startRecord do
20:         set currentRecord := next(currentRecord)
21:         set items := items ∩ currentRecord
22:     return items
23: assignComp(record)
24:     for all i ∈ cycle(twin(record)) do
25:         set i.component := Mn
26:     for all j ∈ cycle(twin(record)) do
27:         if twin(j).comp ≠ ∅ then
28:             assignComp(k)
```

nated record's twin cycle would set their component column (lines 24–25). This is done by using the *twin* function, which returns the twin of that record, and the *cycle* function (lines 16-22), which determines all records in a boundary cycle. The *cycle* function does so by iterating through and collecting records where the halfedge column of the next record equals the current record's next column, until it arrives at the original record. Using record a as an example, the *twin* function would return record $\bar{a}$ and the *cycle* function would return records $\bar{a}$ and c.

After setting the component column, these records are iterated through to see if their twin's component column is empty (lines 26–27). If this is the case, the *assignComp* function is applied to that record (line 28). Following these steps, records would set their component column to M1 in the following order: a $\rightarrow$ $\bar{a}$c $\rightarrow$ $\bar{c}\bar{b}$ $\rightarrow$ ab.

Once this stage of the algorithm has completed for all records in the DCEL table, the component column will then be filled as shown in Table 1. The second stage of the *generateMaptree* function iterates again through all records (line 8). If it finds an edge that is not stored

in the maptree, it will add it to the maptree graph and assign that edge a label (lines 9–10). Using the first record as an example, the edge (0, M1) would be added with the label ab.

### 2.2.2 *Maptree dynamism*

The maptree is based in part on Stell and Worboys' 2011 paper "Relations between adjacency trees" [34], which models dynamic behavior of regions using adjacency trees to represent the configurations of a region at different times. The type of relation between the nodes of two subsequent adjacency trees was termed the bipartite relation, which corresponds to the type of change that occurred to the region. The paper then demonstrated that these bipartite relations are composed of sequences of atomic relations, of which there are four: insert, split, merge, and delete.

While his 2012 paper [28] lays the groundwork for use of the maptree in the description of region dynamism, Worboys' 2013 paper "Using maptrees to characterize topological change" [35] provides a more comprehensive description of this. This paper, like [34], makes use of four atomic relations on nodes of the maptree to describe changes to the region's configuration. These relations are insertion, deletion, unfolding, and folding, and come with corresponding edge label operations.

Insertion occurs when a new white node is added to the maptree and corresponds to the appearance of a region component, whereas deletion corresponds to the disappearance of a region component. Unfolding occurs when a white node splits into two nodes and corresponds to the merging of two region components. Folding occurs when two white nodes merge into a single node and corresponds to the splitting of two region components. These four atomic relations are then used to construct a conceptual neighborhood [23] that corresponds to the RCC (Region connection calculus) [17], although with a greater distinction between the individual relations.

## 2.3 DECENTRALIZED ALGORITHMS

The following section categorizes decentralized algorithms based on their capabilities. Each subsection describes a collection of algorithms capable of meeting the requirements of some aspect of the hypothesis, but are lacking in some other respect. Collectively, the algorithms of this section meet all the requirements needed to test the hypothesis.

### 2.3.1 *Decentralized monitoring of regions*

When representing the structure of areal objects, two methods immediately present themselves: the containment tree, which identifies

strict containment between region components; and adjacency relations, which identify neighboring regions. The containment tree has been shown by Worboys and Bofakos [29] to be computable using the boundaries of region components. Duckham et al. [36] demonstrated that it is possible to compute the containment tree and adjacency relations of different types of region objects using decentralized (in-network) algorithms in a static geosensor network.

Additionally, a particular string of work has developed techniques for detecting topological changes in regions as changes in the containment tree [37–42]. Other work has further extended this approach by adding a third, intermediate category [43] based on the egg-yolk model [15]. This additional category may be useful in applications where there is uncertainty or conflict among the sensors at the boundary of two regions.

The common limitation of all these approaches is that they require the underlying communication graph of the network to be known, to remain constant, and to be plane. A plane graph consists of a planar graph (i.e., a graph that can be drawn with no intersecting edges) and a particular planar embedding of that graph where the edges explicitly do not intersect. To ensure the communication graph is static, the nodes must not move throughout the execution of the algorithm. Further, for a plane communication graph to be generated, the coordinate information for each node must be known.

Recent work has addressed some of these limitations, for example Jeong and Duckham [44] present a collection of decentralized algorithms for computing relations between regions, and Jeong et al. [45] identify critical points (i.e., peaks, pits, and passes), and the topological structure connecting them, of scalar fields. Both of these sets of algorithms are able to perform their task without coordinate information and without the restriction that the communication graph be plane, although they are still restricted to static nodes and regions. Jeong continued this work in his thesis "Qualitative characteristics of fields monitored by a resource-constrained geosensor network" [46], adapting the scalar field algorithms to dynamic fields.

### 2.3.2 *Group movement patterns*

Considerable work has already been completed using centralized approaches to detecting and classifying movement patterns of mobile objects [47–50]. Such movement patterns include flocking [51, 52], convoys [53], and leadership [54–56]. Some work has also been completed using decentralized approaches to movement pattern detection. For example, both flocking [57, 58] and convoys [59] have been detected. While these approaches do account for movement in the nodes, they are only capable of discovering movement patterns and not sensing information about their local environment.

### 2.3.3 *Location-free, mobile node based approaches*

So far, the algorithms that have been introduced either require location information and static networks, or they account for movement in the nodes but are not designed to sense information about their local environment. There are however decentralized algorithms that are capable of providing information about the characteristics of geosensor networks with mobile nodes without relying on coordinate information.

While location information required for geosensor networks is generally taken to mean that the nodes are equipped with location sensors such as GPS, this is not necessarily the case. Decentralized algorithms have been specified that use distance estimation to determine the position of nodes in the network [60–63]. However, these algorithms assume that the nodes are static, although some work has been completed that can account for some node mobility [64, 65]. Based on Nagpal et al.'s work [63], Liu et al. [66, 67] devised an algorithm capable of estimating the distance and therefore position of mobile nodes in a geosensor network. Other work has analyzed the effect of different types of node mobility on these types of algorithms [68].

Work focused on monitoring the health of geosensor networks has been conducted based on the Push-Sum algorithm [69], which spreads a mass value throughout a static network via gossiping. Measuring how this mass value deviates from the expected values is used as a proxy for the amount of communication failures and network "churn" (number of nodes entering and leaving the network). This mass-based approach has been extended to networks with mobile nodes to detect these communication failures [70, 71] as well as network churn [72].

An important related aspect of working with geosensor networks is clustering, which is the partitioning of the network into a series of regions. Some work has been completed to account for node mobility when establishing and maintaining clusters [73, 74]. Using a mass-based approach, Pruteanu and Dulman [75] devised an algorithm that is able to divide a network of mobile nodes into regions and maintain the positions of these regions. However, these regions do not correspond to sensed values, while other approaches take the closeness of sensed values into account when partitioning the network [76]. The key attribute common to these mass based algorithms is that they are not just capable of running on geosensor networks with mobile nodes, but actually rely on this mobility to perform their function. This exploitation of movement through the exchanging of information is known as the mobility diffusion effect [57, 65].

It has been argued that decentralized algorithms are currently capable of detecting structure and changes in the topological relationships of complex areal objects, provided that the nodes are stationary and that the nodes are aware of their location. While some recent work has adapted decentralized algorithms to detect the qualitative characteristics of scalar fields using location-free geosensor networks, the geosensor network must still be static.

It has also been presented that decentralized algorithms exist to monitor movement patterns of mobile nodes, but not the topological relationships of sensed regions. Lastly, decentralized algorithms have been introduced that are capable of providing information about the characteristics of mobile geosensor networks without relying on coordinate information, however they have yet to be applied to sensing external phenomena. In summary, there is a clear gap in the current research for decentralized algorithms that qualitatively analyze and monitor dynamic spatial phenomena using geosensor networks where the nodes are both mobile and coordinate-free.

Therefore, it is the purpose of this thesis to produce decentralized algorithms capable of filling this gap. Based on the specifications of Chapter 3, Chapters 4 and 5 present new decentralized algorithms capable of determining the internal structure of regions and discovering any qualitative relations that may be present.

# 3

# METHODOLOGY

In order to design, implement and test decentralized algorithms capable of fulfilling the research questions outlined in the first chapter, an approach must be formalized. To design decentralized algorithms capable of running on a geosensor network, a formal model must be selected for both the decentralized algorithms and the geosensor network they run on. These decentralized algorithms must then be implemented in an agent-based simulation system, which in this case will be NetLogo [77]. Finally, these algorithms must be evaluated based on the criteria of scalability and veracity.

## 3.1 ALGORITHM DESIGN

### 3.1.1 *Static geosensor network*

As discussed in the previous chapter, geosensor networks are a collection of nodes tasked with monitoring their local environment in geographic space and communicating with their local neighbors. In order to carry out these tasks, nodes in a geosensor network require the following three features; the capability to capture environmental data from sensors, to compute with this data using an on-board microcontroller, and to communicate data using short-range wireless communication [8].

The first feature of a functional geosensor network is that the nodes comprising the network are capable of sensing their environment. Recall from section 2.1.1 that sensors are modeled as functions, i.e., $sense : V \rightarrow C$ where $V$ is the set of nodes in the network and the codomain $C$ is the corresponding value returned by their sensor. Using the geosensor network from Figure 7 as an example, nodes are equipped with a specific sensor, $s$, capable of sensing whether the node is within the grey region or not. Assuming that this senor returns 1 if the node is within the region and 0 if it is not, then this particular sense function can be written as follows; $s : V \rightarrow \{0, 1\}$.

The second feature of a geosensor network, that nodes are capable of performing computations on sensed data, is described using decentralized algorithms, as discussed in the next section. One component that is required for decentralized algorithms is that each node is equipped with a unique identifier. This is achieved with the identifier function $id : V \rightarrow \mathbb{N}$.

The final feature of a geosensor network, that nodes are able to communicate wirelessly, is formally modeled as an undirected graph $G =$

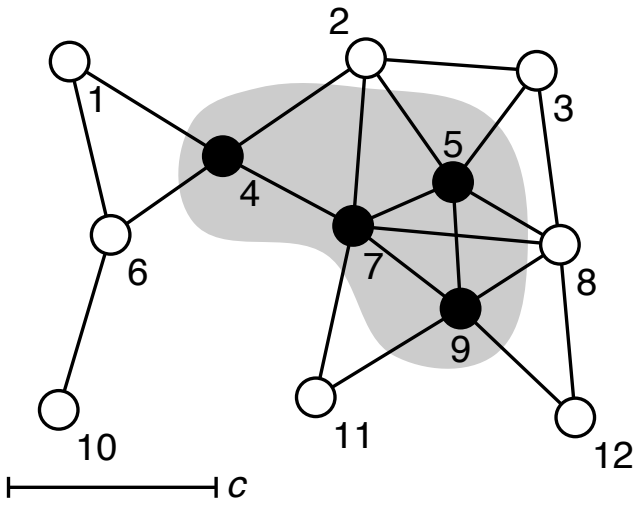



Figure 7: Example geosensor network. Nodes that have sensed the grey region are colored black, while nodes that have not are white. One-hop wireless communication links between nodes are represented as black lines, with a maximum distance of $c$.

$(V, E)$. In this communication graph, $V$ represents the set of nodes comprising the geosensor network and $E$ represents the set of one-hop communication links between those nodes, i.e. $E \subseteq V \times V$. Using the geosensor network from Figure 7 as an example, there is a set of 12 nodes: $V = \{1, 2, \ldots, 12\}$ and 21 edges: $E = \{(1,4), (1,6), \ldots, (9,12)\}$.

As a direct consequence of the communication network, each node must be aware of its neighborhood. This neighborhood, which is the subset of edges from the communication graph incident to that node, is sensed by the *nbr* function. Formally this is written as $nbr : V \to 2^V$, where $nbr(v) \to \{v' \in V \mid (v, v') \in E\}$. Using node 2 from the geosensor network in Figure 7 as an example, there is a set of 4 neighboring nodes: $nbr(2) = \{3, 4, 5, 7\}$.

It is important to note that the communication graph of Figure 7 is neither planar nor plane (i.e., edges in the graph are able to and do intersect without the presence of a node). While some decentralized algorithms require that the communication network be plane, the algorithms described in this research make no such specification. As such, the geosensor networks used will be constructed using the Unit Disk Graph (UDG), where only nodes within a specified distance of $c$ are able to directly communicate [8].

### 3.1.2 *Algorithm specification*

Now that a formal model has been specified for the geosensor networks, decentralized algorithms can be described. The decentralized algorithm specification style used in this research is based on the work developed by Santoro [78] and extended by Duckham [8]. This

specification style was chosen as it is specifically tailored towards the description of decentralized algorithms. This focus on decentralization is in contrast to the majority of styles that assume a centralized system, giving little focus as to the local knowledge of nodes, nor to the communication between these nodes.

This not to say that the chosen style is the only method capable of representing and analyzing decentralized systems. Milner's $\pi$-calculus [79] models the changing connectivity of interactive systems in terms of both communication and computation. $\pi$-calculus, while a powerful set of formalisms that can be applied to decentralized systems, exist at a very low level of abstraction where the degree of fine-grained control over all aspects of the system mean that even simple examples produce verbose and difficult to interpret code.

Like $\pi$-calculus, the asynchronous network algorithms and input-output automata of Lynch [80] produce precise formal descriptions of decentralized systems, although at a higher level of abstraction. Such formal descriptions allow for aspects of decentralized systems to be formally proven. For example, liveness and safety are properties that allow for formal proofs to guarantee that specified outcomes will or will not occur respectively. Similar proofs can also be developed to detect deadlock, where the system enters a cyclic pattern it is unable to recover from.

Such tools, while extremely useful for debugging any decentralized system's design before implementation, come at a cost: requiring the very precise formulation of algorithms in a way that is difficult to scale to more practical algorithm design, which must take into account a wide range of extended programming constructs, such as control flow, uncertainty in sensed data, and node communication. In this way, the specification style of Santoro and Duckham complements these fine-grained tools, providing a higher level of abstraction suitable for the design of algorithms that are closer to a practical implementation, and more suited to simulation and experiments. Additionally, this specification style has the benefit of closely mapping to the implementation of the algorithms, which will be discussed in section 3.2.

The chosen specification consists of four components; restrictions, states, events, and actions. To aid in the explanation of these components, an example algorithm will be used. Consider the geosensor network from Figure 7 and suppose it is necessary for each node in the network to know which nodes are within the shaded region. This can be achieved using a surprise flooding algorithm similar to that found in [8], the specifications of which can be found in algorithm 2. Surprise flooding has been chosen for the example as it serves as the basis of many key components of the algorithms featured in this work.

**Algorithm 2** Surprise flooding algorithm

1: Restrictions: reliable communication; connected, bidirected communication graph $G = (V, E)$; neighborhood function $nbr : V \to 2^V$; sensor function $s : V \to \{0, 1\}$; node identifier function $id : V \to \mathbb{N}$.
2: State transition system: $\langle \{\textsc{idle}\}, \varnothing \rangle$
3: Initialization: All nodes in state IDLE
4: Local variables: Set $S \subset \mathbb{N}$, initialized to $S := \varnothing$

IDLE

5: *Spontaneously*
6: **if** $\mathring{s} = 1$ **then**
7: **broadcast** (msge, $\mathring{id}$)
8: *Receiving* (msge, $id'$)
9: **if** $id' \notin S$ **then**
10: **set** $S := S \cup \{id'\}$
11: **broadcast** (msge, $id'$)

Restrictions define the assumptions made about the geosensor network and are listed in the algorithm's header (line 1). In this case, reliable bidirected communication between the nodes, a neighborhood function to determine the node's neighbors, a sensing function to detect whether a node is within a region, and a node identifier function to uniquely identify nodes are all assumed.

States allow for the retaining of knowledge from previous interactions, allowing nodes to respond differently to the same event based on this information. States are formatted in small capitals (i.e., IDLE) and are described using the state transition system (line 2). The state transition system is a tuple of length two, with the first field listing the set of possible states, and the second field listing the set of possible transitions between these states. While algorithm 2 has a very simple state transition system consisting of a single state and no transitions, module 1 from Chapter 4 utilizes a more complex system, where nodes may transition from the INIT, to the LEAD state, before transitioning to the REGN state.

Events model the interaction between a node and its environment, including interaction with other nodes. There are three types of events that may occur, all of which are formatted in italics (i.e., *Spontaneously*). The first type are trigger events, which occur when a node detects the activation of a trigger. These are indicated by the *When* keyword. While none are present in this algorithm, in the following algorithm this is used to trigger a response when a node's sensed value changes.

The second type are communication events, which occur when a node receives a message from its neighbor. This is indicated with the *Receiving* keyword, such as when a node receives a `msge` message (line 8). Messages are broken up into a sequence of fields, taking the form of the tuple $M = \langle f_1, f_2, \ldots, f_n \rangle$. The first field is always the message type header, and is typeset using a typewriter font (i.e., `msge`). By requiring messages to be typed, nodes are able to respond differently

based on the type of message received. Messages are required to have a fixed number of fields ($n$), which can vary based on the message's type. The later fields then carry data required for the algorithm's computation (e.g., $\mathring{id}$ in line 7).

Lastly, there are spontaneous events, which originate from outside the network. An example of this would be an observer turning on a node. In this algorithm, a spontaneous event occurs when the algorithm begins (line 5), causing nodes that have detected that they are within a region to broadcast a message containing their id.

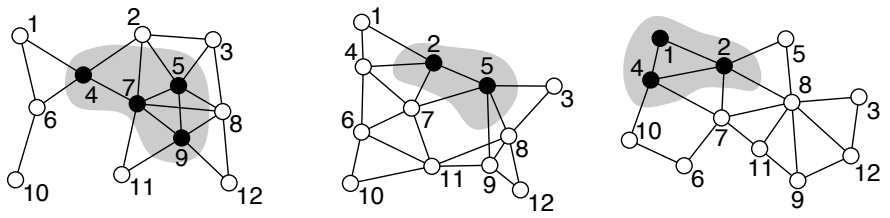
Actions are the sequences of operations that nodes use to respond to events, and are formatted in bold (i.e., **broadcast**). These actions are the traditional "program" aspect of decentralized algorithms and are atomic in nature as they cannot be interrupted by other events, i.e., nodes can only react to one event at a time. In the case of algorithm 2, when a node detects it is within a region, it broadcasts a message containing its id (lines 6–7). When a node receives this message, it checks to see if has already stored the received id in its local storage $S$ (lines 8–9). If it has not, it stores the id and rebroadcasts the message (lines 10–11).

To reduce ambiguity, algorithms in this work make the distinction between global variables and functions, which use no notation (e.g., $id$); the local versions of these functions and variables, which use over-dot notation (e.g., $\mathring{id}$); and received versions of these functions and variables, which use prime notation (e.g., $id'$).

Using the geosensor network in Figure 7 as an example, at the completion of this algorithm each node will have the node ids 4, 5, 7, and 9 stored in its local memory, i.e. $S = \{4, 5, 7, 9\}$.

### 3.1.3 *Dynamic geosensor network*

While the surprise flooding algorithm described previously is capable of determining which nodes are within a region, it must do so with the assumption that the region and nodes are immobile. This is a limitation of static geosensor networks that needs to be addressed. Consider a geosensor network where the nodes and sensed region are mobile, such as that shown in Figure 8.




a. t=100. b. t=200. c. t=300.

Figure 8: Example dynamic geosensor network at three sequential timesteps with mobile grey region.

Given that the communication graph of the geosensor network is based on the UDG and that the nodes are mobile, edges must be added and removed from the communication graph when nodes move within and without the communication distance. For example, in Figure 8, a communication link is added between nodes 6 and 11 at time 200 and removed at time 300.

Representing these changes will require extending the static communication graph $G = (V, E)$ to a time varying communication graph. This graph can be formally represented as $G(t) = (V, E(t))$, where $G(t)$ is the communication graph at time $t$ and $E(t)$ is the set of communication edges at time $t$.

This time varying communication graph can then be used to extend the neighborhood function *nbr*. Formally this is written as $nbr : V \times T \to 2^V$, where $nbr(v, t) \to \{v' \in V \mid (v, v') \in E(t)\}$. Using node 10 from the geosensor network in Figure 8 as an example, there are three different sets of neighboring nodes: $nbr(10, 100) = \{6\}$, $nbr(10, 200) = \{6, 11\}$, and $nbr(10, 300) = \{4, 6\}$.

The sense function can also be extended to account for dynamism. Again, assuming that the senor returns 1 if the node is within the region and 0 if it is not, then the sense function for any given time can be written as $s(t) : V \times T \to \{0, 1\}$. Using node 4 from Figure 8 as an example, this would give three different values: $s(100) = 1$, $s(200) = 0$, and $s(300) = 1$. By defining a time varying sense function the nodes are able to account for both node movement and region movement.

While the surprise flooding algorithm described previously is capable of determining which nodes are within a region for a static geosensor network and region, this would be insufficient for the geosensor network shown by Figure 8. Alternatively, the algorithm could record the times a node enters or exits the region. It would then be a simple task of querying these records to determine how many nodes are within the region at any given time.

**Algorithm 3** Surprise flooding algorithm for dynamic networks

1: Restrictions: reliable communication; connected, bidirected communication graph $G(t) = (V, E(t))$; neighborhood function $nbr : V \times T \to 2^V$; sensor function $s : V \times T \to \{0, 1\}$
2: State transition system: $\langle \{\text{IDLE}\}, \varnothing \rangle$
3: Initialization: All nodes in state IDLE
4: Local variables: Set $S \subset \{0, 1\} \times T \times \mathbb{N}$, initialized to $S := \varnothing$

IDLE

5: *When* $\mathring{s}(now)$ *changes*
6: **broadcast** (msge, $\mathring{s}(now)$, *now*, $\mathring{id}$)
7: *Receiving* (msge, $s'$, $t'$, $id'$)
8: **if** $(s', t', id') \notin S$ **then**
9: **set** $S := S \cup \{(s', t', id')\}$
10: **broadcast** (msge, $s'$, $t'$, $id'$)

The specifications of this algorithm and geosensor network are shown in algorithm 3, which has extended its restrictions and local storage (lines 1 and 4) to handle mobility in both the nodes and the sensed region. In this algorithm, nodes that change their sensed value (either through moving in/out of the sensed region or the region moving towards/away from the node) broadcast a message to their neighbors with their new sensed value, the current time and their id (lines 5–6). Nodes receiving this message will, as previously, check to see if it has already stored the received record (lines 7–8) and if it hasn't, stores the record and rebroadcasts the message (lines 9–10).

Of note in Figure 8 and algorithm 3 is that while the nodes may change position, they are not aware of their own position. While some decentralized algorithms require a position function for their nodes (i.e., $p : V \times T \to \mathbb{R}^2$), the algorithms described in this research make no such restriction.

Using the geosensor network in Figure 8 as an example, at the completion of this algorithm each node will have the following stored in its local memory:

$S = \{(0,100,1), (1,300,1), (0,100,2), (1,200,2), (0,100,3), (1,100,4), (0,200,4), (1,300,4), (1,100,5), (0,300,5), (0,100,6), (1,100,7), (0,200,7), (0,100,8), (1,100,9), (0,200,9), (0,100,10), (0,100,11), (0,100,12)\}$.

Using this data, it can be deduced that there were four nodes within the region at time period 100 $(4,5,7,9)$, two nodes at time period 200 $(2,5)$, and three nodes at time period 300 $(1,2,4)$.

## 3.2 ALGORITHM SIMULATION

Now that a formal model for specifying geosensor networks and the decentralized algorithms that run on them has been chosen, it is now possible to implement these algorithms on simulated geosensor networks. Simulated networks, as opposed to implementation on actual hardware and field deployment, have been chosen for this work as it is quicker and easier to implement the specifics of node hardware and the communication protocols can be abstracted.

While there are several programs available to simulate sensor networks (e.g., ns-3, OMNet++, TOSSIM), their development is focused on the specifics of node hardware and communication protocols. For this reason, agent based modeling (ABM) systems will be used instead. ABM systems consist of a set of autonomous agents within a simulated environment. This research maps well to this framework, with agents acting as geosensor nodes and the simulated environment representing the geographic environment. Some examples of ABM systems are NetLogo, MASON, Repast, and Swarm. Out of these options NetLogo will be used in this research.

### 3.2.1 *NetLogo simulation environment*

The NetLogo simulation environment [77] was chosen for this project as it is an actively developed, cross-platform, and open-source simulation system that is well suited for the modeling of complex multi-agent systems. Of particular value is its ease of use when programming as the coding style maps very closely with the algorithm specification style. Figure 9 shows how closely the implementation of algorithm 3 in NetLogo matches that of the algorithm specification.

```
to go
  ask nodes [
    node_movement_procedures
    if state = "IDLE" [step_IDLE stop]
  ]
  simulation_visualization_procedures
  tick
end

to step_IDLE
  if s != s_old [ ;; When s(now) changes
    broadcast (list "msge" s ticks who)
  ]
  while [has-message "msge"] [
    ;; Receiving message (msge, s', t', id')
    let message received "msge"
    let s' item 1 message
    let t' item 2 message
    let id' item 3 message
    if (member? (list s' t' id') Storage = false) [
      set Storage lput (list s' t' id') Storage
      broadcast (list "msge" s' t' id')
    ]
  ]
end
```

Figure 9: NetLogo code for algorithm 3.

Before discussing the specifics of any code, it is important to first discuss the way in which NetLogo simulates an algorithm. NetLogo uses a pseudorandom number generator where random seeds are generated at the start of each simulation based on the current date and time. It is important to note that all random and scheduling functions draw from this generator, meaning that every simulation is entirely deterministic. A deterministic approach to simulation allows for reproducibility when assigning a specific random seed, and is particularly useful when debugging code. Figure 10 shows a sim-

ulation of algorithm 3 on a geosensor network with 200 nodes. The area used for simulating the environment is defined by a two dimensional grid of discrete square cells, called patches, with arbitrary dimensions. These patches can be assigned attributes, and in the case of Figure 10, there is a grid of 640×480 patches with a region attribute coloring some patches grey.

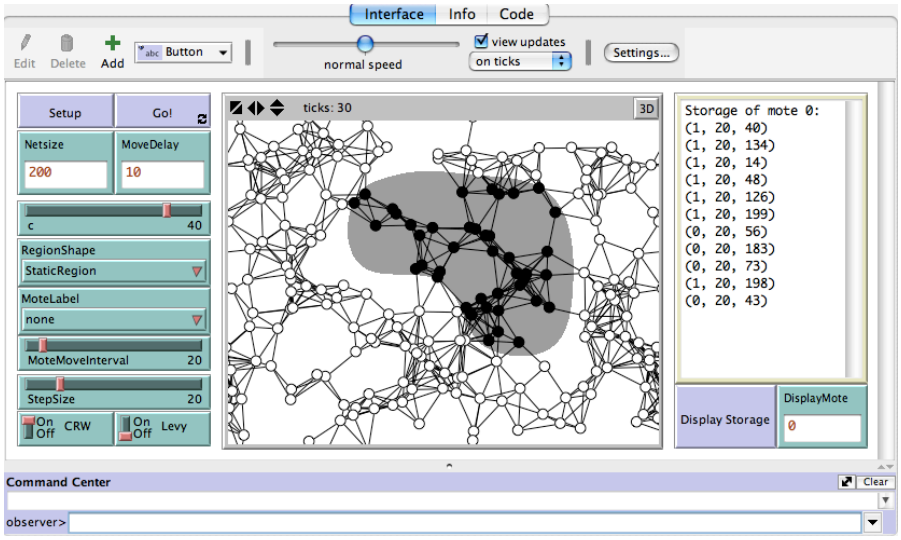



Figure 10: Screenshot of the Netlogo simulation environment running algorithm 3 showing the world simulator (center), simulation controls (left and top), and simulation output (bottom and right).

The primary type of agents in NetLogo's simulation system are referred to as turtles and are capable of running their own code in addition to interacting with their environment and each other. In this work, these turtles are re-purposed as mobile geosensor nodes and are shown as black or white circles depending on their sensed value in Figure 10. Every simulation begins by placing a specified number of nodes within the world at random locations and with random headings. Simulating communication between nodes is done using the links agent type. These communication links are represented with black lines between nodes that are within a specified communication range, which in the case of this example is 40 patch widths.

The unit NetLogo uses to measure time is called a tick and represents the amount of time used to execute one complete run-through of the simulation procedures. The speed of these ticks can be altered using the simulation controls at the top of the window. Clicking the go button activates the go procedure, which first asks the nodes to move and update their sensed value. The nodes then run a specific procedure determined by their current state, which in this algorithm is always idle. After this, some visualization procedures are run to ensure that the nodes are colored according to their sensed values before finally bringing forward the simulation one time step. The rest

of the code then functions identically to that of the specifications of the algorithm.

A powerful feature of NetLogo is the ability to read the outputs of the agents (i.e., the turtles, links and patches). This output can also be scripted, with the simulation of Figure 10 outputting the storage of node 0.

### 3.2.2 *BehaviorSpace*

When evaluating decentralized algorithms on geosensor networks, it is necessary to vary simulation parameters and record the results in order to have a detailed understanding of the algorithm's effectiveness. NetLogo's BehaviorSpace tool is capable of automating this, allowing simulations to be run many times under varying parameters. BehaviorSpace then records the specified outputs to a csv (comma separated value) file. A key feature of BehaviorSpace is that each run of the experiment is executed independently, meaning that the simulation can assign individual runs to separate processor cores.

To give an example of such an experiment, suppose it is necessary to determine how the amount of messages sent by algorithm 3 increases with the size of the geosensor network. This would involve varying the network size and recording the amount of messages sent after a specified time period has elapsed. An output of this can be seen in Figure 11 where the network sizes of 100, 200, 300, 400, and 500 were chosen. This experiment has been run 100 times for each network size with a region shape identical to that of Figure 10. Messages were measured after 500 time steps and nodes were able to move 10 times.

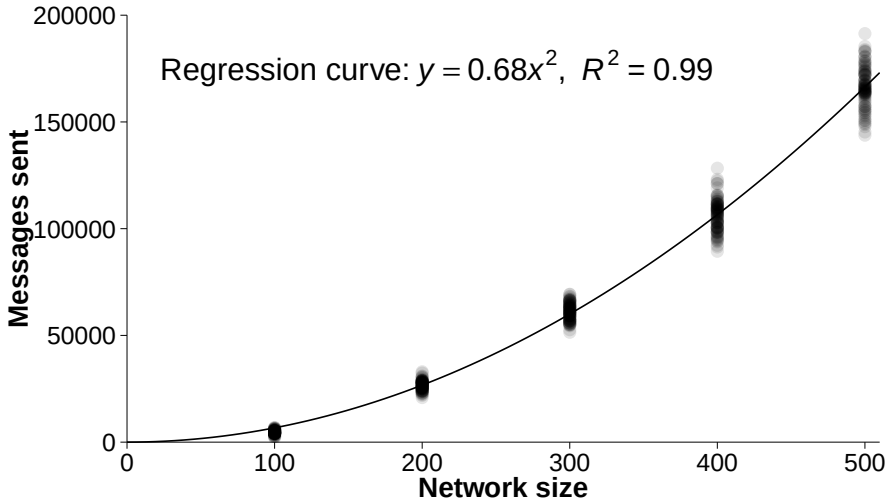



Figure 11: Graph plotting the relationship between network size and number of messages sent by the network with fitted curve.

Using this data a curve can be fitted using regression analysis to determine the precise relationship between the size of the geosensor network and the amount of messages sent by the nodes. From this graph, it can be observed that this relationship is polynomial.

## 3.3 ALGORITHM EVALUATION

Now that it is possible to construct experiments to determine the interactions between simulation parameters, the decentralized algorithms can be formally evaluated. The ways in which decentralized algorithms are evaluated can be divided into two categories: scalability and veracity.

### 3.3.1 *Scalability*

When evaluating scalability, the primary concern is how the amount of communication is affected by an increase in network size. This is because the nodes will have limited battery supply and wireless communication uses by far the most energy of any of the nodes' systems. Considering the amount of communication necessary for the entire network as a function of network size is known as communication complexity, whereas the amount of communication for individual nodes is known as load balance.

*Communication complexity*

This work will use big-oh notation to categorize the algorithms into orders of scalability. Specifically, write $O(f(n))$ (where $n$ is the network size) to mean that the amount of communication has an upper bound of some constant of that function. For example the communication complexity of algorithm 3 is $0.68n^2$, which in big-oh notation would be $O(n^k)$. Figure 12 shows the different classes of functions that are possible, listed in order of decreasing efficiency.

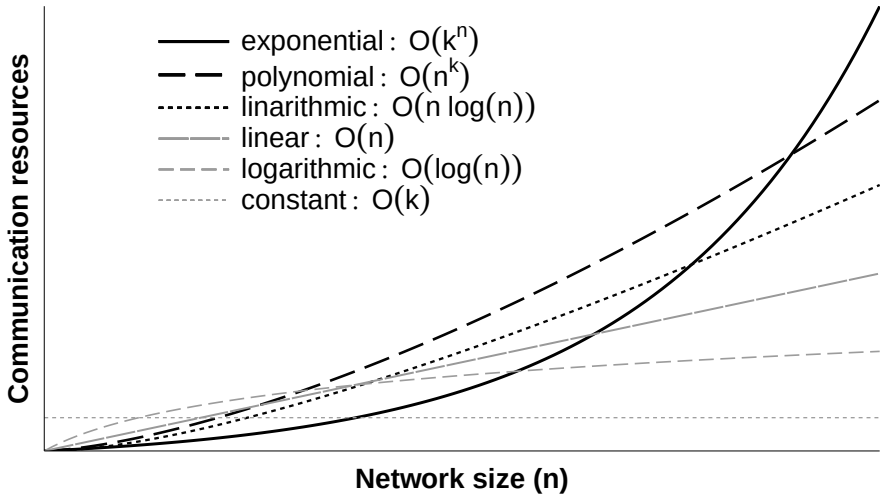



Figure 12: Typical response curves for communication complexity, after [8].

These orders of scalability can be used to evaluate the relative efficiencies of related decentralized algorithms. For example, Figure 13 illustrates that algorithms 1 and 2 have linear orders of scalability, $O(n)$, whereas algorithm 3 has the less efficient polynomial order of scalability, $O(n^k)$.

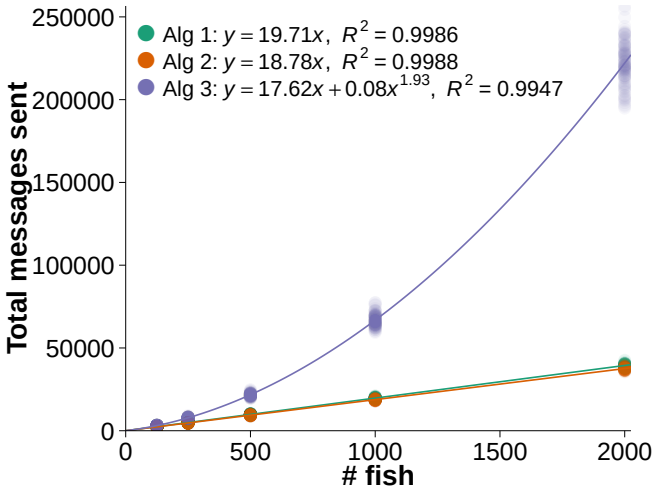



Figure 13: Scalability of communication in terms of total number of messages sent with change in numbers of fish for Algorithms 1—3, adapted from [9].

This graph originates from prior work on decentralized monitoring of moving objects in transportation networks [9]. In this paper, three algorithms are utilized that run at each cordon of a cordon-structured network (a transport network augmented with checkpoints capable of logging the passing of moving objects) to record the time periods that agents (referred to as fish) are on adjacent edges. Algorithm 1 assumes that cordons sharing an edge are able to communicate with

each other whereas algorithm 2 assumes that cordons are unable to directly communicate with each other and must use the agents to ferry the data between cordons. As it takes time to ferry the data, record completion is delayed. Algorithm 3 extends algorithm 2 by allowing agents to communicate with each other on the edges, reducing the delay in record completion.

As the number of cordons is fixed, this leads to linear scalability for algorithms 1 and 2 as communication only occurs when fish pass cordons. The reason for algorithm 3's less efficient communication complexity is due to its additional requirement of fish-fish communication. As increasing the number of fish also increases the number of fish-fish communication events, this leads to a worst case scenario of the communication complexity scales with the square of the total number of fish.

*Load balance*

In addition to testing the scalability of the entire network, scalability can also be evaluated based on the amount of communication of individual nodes. This is known as load balance. Like the previous graph, Figure 14 originates from decentralized monitoring of moving objects in transportation networks. This work additionally tested for load balance, which was calculated using the maximum communication load of any node in the network to determine the worst-case communication complexity. In the case of this paper, all three algorithms exhibit linear load balance, $O(n)$.

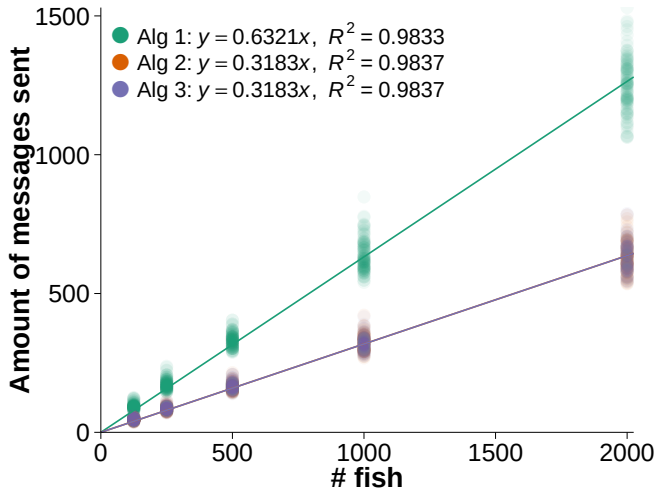



Figure 14: Scalability of communication in terms of worst case (maximum) load (number of messages sent) for any node during a run of Algorithms 1—3, adapted from [9].

### 3.3.2 *Veracity*

Veracity tests the accuracy of decentralized algorithms by comparing the results of the algorithms with the actual results as determined by the environment. An example of this is shown in Figure 15 where two types of nodes (fearful and aggressive) are tasked with identifying the number of regions in their local environment. In this case there are two actual regions that the nodes initially misidentify as up to six regions. Over time, the nodes become progressively more accurate before eventually determining that there are two regions.

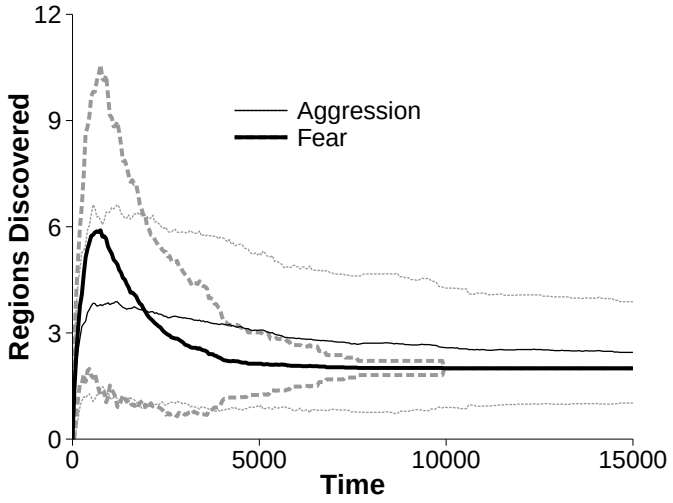



Figure 15: Average number of region objects known to vehicles over simulation time, for total 200 vehicles (100 "fear" type-2a and 100 "aggression" type-2b). The actual number of region objects embedded in the environment is 2 for these experiments, adapted from [10].

Figure 15 originates from work on Spatiotemporal Braitenberg vehicles (SBVs), which are autonomous vehicles that are able to exhibit increasingly complex spatial behaviors. This work produced four classes of SBVs, the third of which was capable of identifying objects such as spatial regions. It did so by constructing knowledge of routes between places to build up knowledge about the neighborhoods that connect known places. By additionally sensing which places are a part of an object, this vehicle was able to determine the number of objects in its environment.

Figure 15 shows an aggregate of 100 experimental runs with vehicles with aggressive and fearful movement types navigating a region with two objects. Initially, the vehicles have no knowledge of any region objects but as they navigate their environment and encounter parts of the regions, their estimate of the total number of regions increases rapidly past the actual number of regions. As the vehicles

continue, these initially separate regions merge. Over time, both vehicle types tend towards accurate knowledge of the number of regions.

## 3.4 SUMMARY

Nodes in a geosensor network have the capability to capture environmental data from sensors, to compute with this data using an onboard microcontroller, and to communicate data using short-range wireless communication. Algorithms featured in this research are designed to operate on geosensor networks with the following capabilities:

- Nodes may move;
- Nodes are uniquely identifiable (i.e., identifier function; $id : V \to \mathbb{N}$);
- Nodes have no access to coordinate positioning information;
- Nodes are able to communicate with other nodes (i.e., communication graph $G(t) = (V, E(t))$) within a fixed distance, which is considerably shorter than the span of the network;
- Nodes are aware of their current neighbors (i.e., neighbor function $nbr : V \times T \to 2^V$); and
- Nodes' sensors are able to detect whether they are currently over a positive or negative region component (i.e., sensor function $s : V \times T \to \{0, 1\}$).

Algorithms are written according to the specifications developed by Santoro [78], extended in [8] and then implemented using the NetLogo simulation environment [77]. Algorithms are then evaluated according to scalability, veracity, and load balance.

# 4

# STATIC REGIONS

This chapter presents three decentralized algorithms that are capable of determining the internal structure of static regions and discovering any qualitative relations that may be present. The first algorithm describes the calculation of the containment relationships between region components and the adjacency relations between Voronoi regions of those components. The second algorithm introduces the simplified maptree formal model and data structure for the storage of both containment and Voronoi-adjacency relations for simple regions. The third algorithm extends this to allow for complex regions.

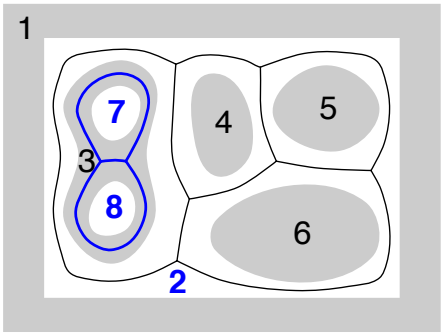
## 4.1 BASIC DATA STRUCTURE

By describing the internal structure of the region as a complex areal object, the individual region components can be modeled as a series of positive regions (islands) and negative regions (holes). The containment relations between these components can be described using a containment tree [29], with the unbounded exterior region serving as the root of the containment tree. This qualitative spatial relation between region components provides the notion of topological distance from the exterior region. Figure 16 shows an example of how exterior region 1 contains hole 2, which in turn contains islands 3, 4, 5, 6, and island 3 contains holes 7 and 8.

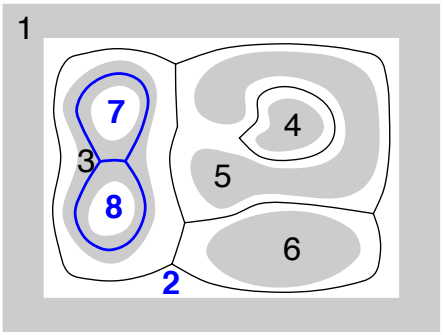
In addition to the containment relation, the adjacency relation is used to further describe the internal structure of the region. Like containment, adjacency describes a qualitative spatial relation between region components. Typically, adjacency is used to describe a pair of regions that directly border each other. For example, in Figure 16, the positive region component 3 is adjacent to the negative region components 2, 7, and 8. Looking at Figure 16's containment tree, these direct adjacency relations are all present as edges in the containment tree, making their storage redundant.

Instead of these direct adjacency relations, the adjacency relations of the Voronoi regions generated by the boundaries of region components will be used. Specifically, the adjacency relations between Voronoi regions enable refinements in the qualitative structure of a complex areal object. For example, Figure 16 shows three different configurations of a complex areal object that has identical containment trees.

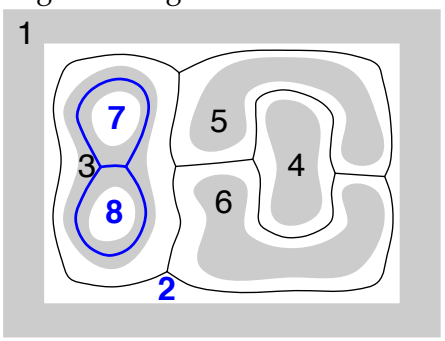
These adjacency relationships between the Voronoi regions generated by the positive and negative region components (black and blue,



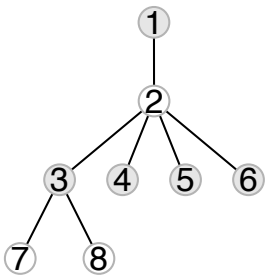

Region configuration a.




Region configuration b.




Region configuration c.




Containment tree.

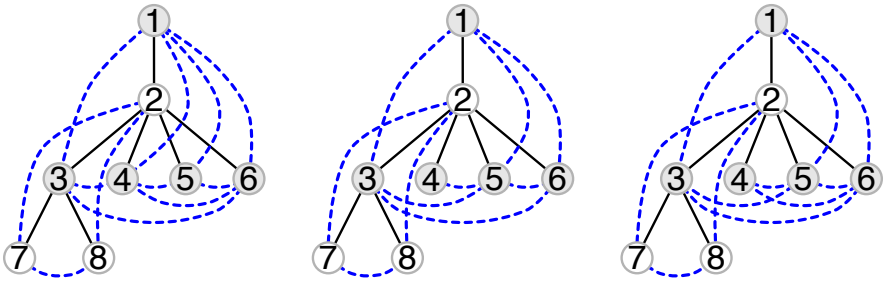
Figure 16: Example complex areal objects with identical containment trees. Positive region components are grey and negative region components are white. Black lines and thick blue lines represent the Voronoi boundaries induced by the positive and negative region components respectively.

respectively) can be used to distinguish these configurations. For example, the Voronoi region generated by positive region 4 has different adjacency relationships in each of the three subfigures in Figure 16. Combining the information from the containment tree and the adjacency relations of the Voronoi regions of positive region components yields three different structures, summarized in Figure 17.

### 4.1.1 *Qualitative relations*

Looking at the combined structure in Figure 17, it can be seen that regions 4, 5, and 6 exist in a different configuration for each of the examples. With reference to Figure 16, it could be said intuitively that region 4 is *surrounded* by regions 5 and 6 in Figure 16c, *surrounded* by region 5 in 16b, and *not surrounded* by any region in 16a.

More precisely, consider a containment tree $C$. For a particular region component $c \in C$, $level(c)$ is written to refer to its level in the containment tree. The root $r$ of $C$ is defined to be at level one, $level(r) = 1$. Node level can be used to partition $C$ into positive and negative nodes. For example, assuming the root of the tree is a positive region (as in




Configuration a. Configuration b. Configuration c.

Figure 17: Combined containment tree and Voronoi-adjacency relations of regions from Figure 16. Containment edges are black and Voronoi-adjacency edges are blue and dashed.

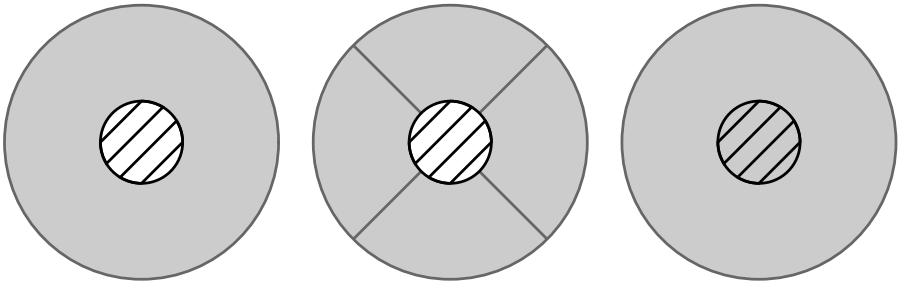
Figure 16), then the set of positive nodes is $\{p \in C | level(p) \text{ is odd}\}$. Without loss of generality, assume $R \subset C$ is the set of all positive region components in $C$. The set of Voronoi regions of the boundaries of these region components is written $V(R)$. The adjacency relation on $V(R)$ is then a symmetric relation $A \subseteq V(R) \times V(R)$. Finally, a distance function is defined $\delta : R \times R \to \mathbb{N}^+$ on $A$, such that $\delta(c, c')$ is the length of the shortest path through $A$ from $c$ to $c'$ (i.e., $\delta(c, c') \to 0$ if $c = c'$; $\delta(c, c') \to 1$ if $(c, c') \in A$; $\delta(c, c') \to 2$ if $(c, c''), (c'', c') \in A$ and $(c, c') \notin A$; and so forth). Using these functions, it is now possible to define the surrounds relation as follows:

**Definition** Consider a region component $c \in R$ such that for all $(c, c') \in A$, $level(c) \leq level(c')$. Then, a region component $s \in R$ is said to *partially surround* $c$ iff: a. $(s, c) \in A$; and b. for some $r \in R$ such that $level(r) < level(s)$ then $\delta(s, r) = \delta(c, r) - 1$. Given the set of all region components $S$ that *partially* surround $p$, $S$ *surrounds* $p$.

Informally, a region component $s$ partially surrounds a region $c$ if $c$ is Voronoi-adjacent to $s$, and $s$ is closer to its containing region component than $c$ by exactly one step in the shortest path through $A$. This surrounds relation provides the ability to discern more detail about the configuration of complex areal objects, such as those in Figure 16. It is contended that this relation also accords with human intuition about "surrounding."

This definition has similarities with Dube and Egenhofer's 2014 paper "Surrounds in Partitions" [11], in which surrounds is defined as a topological relation between an inner region and an outer region (or set of regions) that completely encircles the inner region. For this relation to occur, the outer region requires a hole in which the inner region is located. This requirement is in contrast to the contains relation from the 9-intersection model as the inner and outer regions do not share an interior, as illustrated in Figure 18.

The paper presented four types of surrounds relations, with *surroundsAttach* being the relation of interest to this work. The *surround-*



a. Surrounded by a single region.

b. Surrounded by multiple regions.

c. Contained by a single region.

Figure 18: Example diagram showing the topological relations between black striped inner and shaded grey outer regions discussed in [11].

*sAttach* relation requires that the boundary of the inner and outer regions are completely connected, i.e., the regions are attached. Applying this definition to Voronoi regions then produces the same result as the surround relation presented in this work.

### 4.1.2 *Algorithm design*

With the definitions for containment, Voronoi-adjacency, and surrounds relations in hand, the algorithms can now be implemented in pseudocode. The first algorithm will be referred to as the basicStatic algorithm. For simplicity, this algorithm has been broken up into five modules based on function, with the relations between these modules shown in Figure 19. In brief, the five modules work as follows:

1. **Region identification:** Initialize the network using leader election to select and distribute unique ids for each positive and negative region component in the network.

2. **Voronoi region identification:** Using hop count flooding, determine the approximate Voronoi boundaries between positive region components and between negative region components. Nodes store the hop count to each adjacent region component in their boundary table ($B_t$), with Voronoi boundaries lying at nodes where the hop counts from neighboring regions are equal. As the nodes are mobile, this must be refreshed periodically.

3. **Adjacency relation identification:** Using surprise flooding, propagate the Voronoi-adjacency relations to all nodes within a region component. Using the boundary table from module 2, nodes that are on a Voronoi boundary will add this to their Voronoi-adjacency table ($A_t$) and broadcast this adjacency pair.

4. **Containment tree propagation:** Using surprise flooding from a node on the outermost region of the object, generate a segment

of the containment tree for each region (as *parent* and *children*) as well as detect any surrounded regions and add them to the surrounds table ($S_t$).

5. **Node movement:** Whenever a node moves to a different region, have that node request information from neighbors in the new region in order to keep the information obtained from modules 2–4 current.

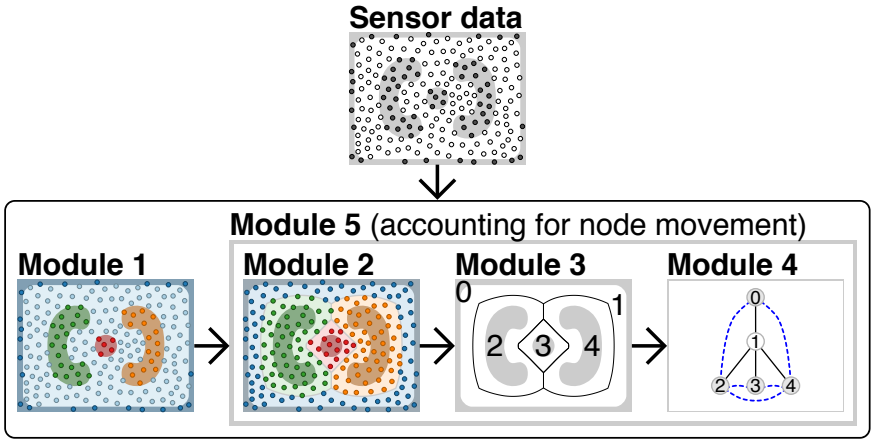



Figure 19: Flow diagram representing the interactions between the modules that comprise the basicStatic algorithm.

### 4.1.3 *Module 1: Region identification*

The network is initialized using leader election [8, 78, 81] to select a unique identifier for each (positive or negative) region component in the network and to distribute that identifier to each node within that region component. Each node begins by setting its node id as its region id (*rid*), then broadcasting a `lead` message containing that id along with its sensed value to its neighbors before transitioning to the LEAD state (lines 5–7). Nodes receiving this message with the same sensed value (i.e., nodes within the same region component) will either set the received id as the new region id and rebroadcast the message if the id is lower; or just rebroadcast the message if their region id has been set to -1 (lines 10–15).

Upon completion, the lowest node id in each region component will be set as the region id for every node within that region component. Precisely when this will occur depends on the diameter of the communication graph of the nodes that make up the region component (i.e, the maximum number of relays it would take a message sent from any node in the region component to reach any other node). This value will be set as the value of the region timer and will differ depending on node density, communication distance, and region

configuration. When this region timer elapses, the nodes will then transition to the REGN state (lines 16–17). Because nodes running this module are mobile, they may change regions before the region timer has elapsed. When this occurs, the node sets its region id to -1 (Lines 8–9). This is to ensure that two adjacent regions are not assigned the same region id.

**Module 1** Region identification

1: Restrictions: reliable communication; connected, bidirected communication graph $G(t) = (V, E(t))$; neighborhood function $nbr : V \times T \to 2^V$; sensor function $s : V \times T \to \{0, 1\}$
2: State transition system: $\langle \{\text{INIT, LEAD, REGN}\}, \{(\text{INIT, LEAD}), (\text{LEAD, REGN})\}\rangle$
3: Initialization: All nodes in state INIT
4: Local variables: region id $rid : V \to \mathbb{N} \cup \{-1\}$, initialized to $\mathring{rid} := \mathring{id}$, sensor function $s : V \to \{0, 1\}$

INIT
5: *Spontaneously*
6: **broadcast** (`lead`, $\mathring{id}$, $\mathring{s}(now)$)
7: **become** LEAD

LEAD
8: *When* $\mathring{s}(now)$ *changes*
9: **set** $\mathring{rid} := -1$
10: *Receiving* (`lead`, $rid'$, $s'$)
11: **if** $s' = \mathring{s}(now)$ AND $rid' < \mathring{rid}$ **then**
12: **set** $\mathring{rid} := rid'$
13: **broadcast** (`lead`, $rid'$, $s'$)
14: **if** $s' = \mathring{s}(now)$ AND $\mathring{rid} = -1$ **then**
15: **broadcast** (`lead`, $rid'$, $s'$)
16: *When* region timer elapsed
17: **become** REGN

### 4.1.4 *Module 2: Voronoi region generation*

Module 2 uses hop-count flooding [8] to determine the approximate Voronoi boundaries between positive region components and between negative region components. Hop count flooding works by incrementing a number every time a message is passed on. By having nodes on the boundaries of region component broadcasting these messages, nodes that are the same amount of hops from two regions can be said to be equidistant between these regions (i.e., on the boundary between these Voronoi region components).

For a network with static nodes, it would be sufficient to send out only one hop message to determine the boundary between Voronoi regions. However, as the nodes within the network are mobile, any Voronoi boundary they determine will become increasingly inaccurate over time and therefore must be periodically refreshed. This refresh interval is handled by the broadcast timer. When this timer

elapses, the timer will be reset, nodes will clear their boundary table ($B_t$), and nodes with a valid *rid* will broadcast a `hop` message containing their sensed value and *rid* along with a hop count of 1 (lines 2–6).

Nodes receiving a `hop` message will first check to see if the received sensed value ($s'$) matches their own. Valid hop messages will either have a differing sensed value and a hop count of 1 (i.e., the message has come from the boundary of an adjacent region component) or the same sensed value and hop count of greater than 1 (i.e, from the same region component but not generated in that region) (lines 7–8). If the node has not received a message from this region component before ($bid'$), it will store the record in the boundary table ($B_t$) before incrementing the hop count and rebroadcasting the message (lines 9–11). Nodes that have received a message from this region component before will update their record if the hop count is lower before incrementing the hop count and rebroadcasting the message (lines 12–15). Lines 16–17 are related to node movement and will be discussed in module 5.

**Module 2** Voronoi region generation

1: Local variables: adjacent region id $adj : V \to \mathbb{N} \cup \{-1\}$, initialized to $\mathring{adj} := -1$, boundary table $B_t = \langle bid : \mathbb{N}, h : \mathbb{N} \rangle$, initialized with zero records.

REGN
2: *When* broadcast timer elapsed
3: Reset broadcast timer
4: DELETE FROM $B_t$
5: **if** $\mathring{rid} \neq -1$ **then**
6: **broadcast** (`hop`, $\mathring{s}(now)$, $\mathring{rid}$, 1)
7: *Receiving* (`hop`, $s'$, $bid'$, $h'$)
8: **if** ($\mathring{s}(now) \neq s'$ AND $h' = 1$) OR ($\mathring{s}(now) = s'$ AND $h' > 1$) **then**
9: **if** (COUNT * FROM $B_t$ WHERE $bid = bid'$) = 0 **then**
10: INSERT INTO $B_t$ VALUES ($bid'$,$h'$)
11: **broadcast** (`hop`, $\mathring{s}(now)$, $bid'$, $(h' + 1)$)
12: **else**
13: **if** $h' < h$ FROM $B_t$ WHERE $bid = bid'$ **then**
14: UPDATE $B_t$ SET $h = h'$ WHERE $bid = bid'$
15: **broadcast** (`hop`, $\mathring{s}(now)$, $bid'$, $(h' + 1)$)
16: **if** $\mathring{s}(now) = s'$ AND $\mathring{rid} = -1$ AND $h' = 1$ **then**
17: **set** $\mathring{rid} := bid'$
18: *Spontaneously* (updating adjacent region id)
19: **let** $MinHop$ := SELECT MIN $(h)$ FROM $B_t$
20: **if** (COUNT $(*)$ FROM $B_t$ WHERE $h = MinHop$) $> 1$ **then**
21: **set** $\mathring{adj} := -1$
22: **else**
23: **set** $\mathring{adj}$ := SELECT $bid$ FROM $B_t$ WHERE $h = MinHop$

The variable *adj* is used to store the id of the region component of the Voronoi region the node is within. For nodes within negative

region components, this will mean the id of the closest positive region component. For nodes within positive region components, this will mean the id of the closest negative region component. This value is selected from a record from the boundary table with the lowest hop count (lines 18–19). Nodes with more than one record with the lowest hop count (i.e., nodes that lay on a boundary between Voronoi region components), set the value to -1 (lines 20–21) whereas nodes with a single record will set the value to the id of that record (lines 22–23).

### 4.1.5 *Module 3: Adjacency relation identification*

Using surprise flooding, when the adjacency timer elapses module 3 propagates the Voronoi-adjacency relations to all nodes within a region component. For module 3 to work, this adjacency timer must be set so that the module only runs after at least one round of module 2 has run. Using the boundary table ($B_t$) from module 2, nodes that have determined that they are on a boundary between adjacent Voronoi regions (i.e., nodes with an *adj* value of -1) will add this record to their Voronoi-adjacency table ($A_t$) and broadcast a `bndy` message containing this adjacency pair along with their sensed value to their neighbors (lines 3–5, 7–8). As module 3 is dependent on information from module 2, module 2 needs to have been run at least once. To ensure that this is the case the adjacency timer must be set accordingly (line 2). Rarely, nodes will find themselves on the boundary between three or more Voronoi region components. However, cases such as these can safely be discarded as other adjacent nodes will be sufficient to represent these boundary pairs (line 6).

**Module 3** Adjacency graph propagation

1: Local variables: Voronoi-adjacency table $A_t = \langle rid_a : \mathbb{N}, rid_b : \mathbb{N} \rangle$, initialized with zero records.

REGN
2: *When* adjacency timer elapsed
3: **if** $\mathring{adj} = -1$ **then**
4: **let** $MinHop$ := SELECT MIN $(h)$ FROM $B_t$
5: **let** *closest* := SELECT *bid* FROM $B_t$ WHERE $h = MinHop$
6: **if** $|closest| = 2$ **then**
7: INSERT INTO $A_t$ VALUES *closest*
8: **broadcast** (`bndy`, $\mathring{s}(now)$, *closest*)
9: *Receiving* (`bndy`, $s'$, *pair*)
10: **if** $\mathring{s}(now) = s'$ AND *pair* $\notin A_t$ **then**
11: INSERT INTO $A_t$ VALUES *pair*
12: **broadcast** (`bndy`, $s'$, *pair*)

Nodes receiving a `bndy` message will check to see if they have the same sensed value (i.e., within the same region) and will only proceed if they do (lines 9–10). This is to ensure that adjacency relations are

kept within the region component that generated them. If this is the case, and they have not already received or generated this adjacency pair, they will add it to their Voronoi-adjacency table and rebroadcast the message (lines 11–12).

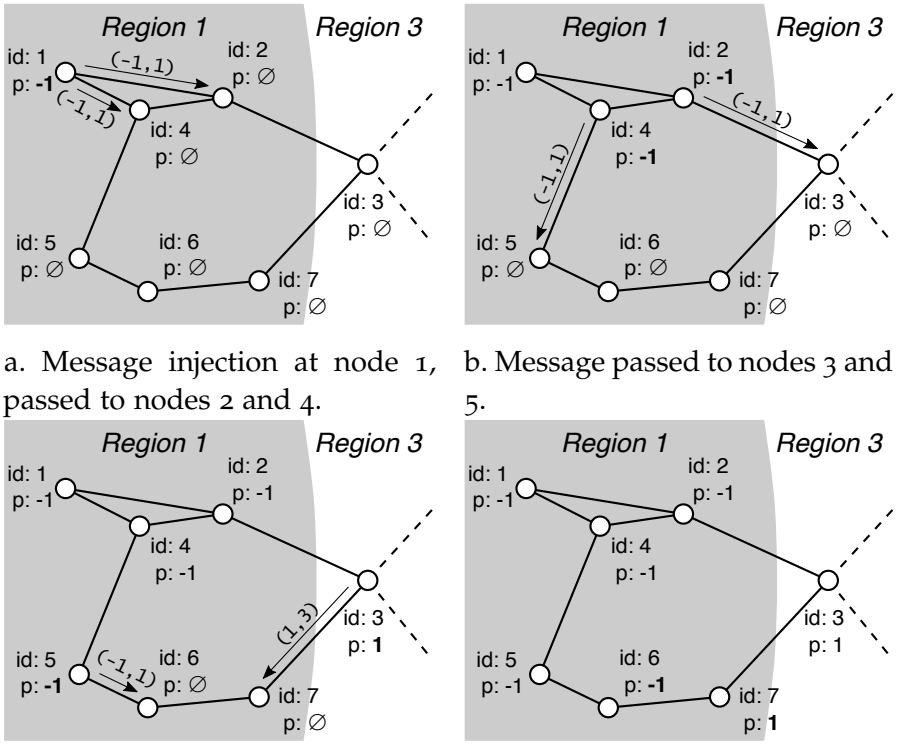
### 4.1.6 *Module 4: Containment tree generation*

This module uses surprise flooding from a node on the outermost region of the object to generate a segment of the containment tree for each region (by populating the *parent* variable and *children* set) as well as detect any surrounded regions and add them to the surrounds table ($S_t$). For module 4 to work, this message must be injected after module 3 has run. From modules 1 to 3, each node now knows the id of the region component it is in (*rid*), the id of region components adjacent to that component along with their relative distances ($B_t$), and the Voronoi adjacencies of those region components ($A_t$).

Each node is then tasked with the generation of the local segment of the containment tree (i.e., determining the region's *parent* and *children* regions) as well as any surrounds relations between these regions. Recall that the rule for detecting surrounded regions is that a region is considered surrounded if there exists no direct link in the adjacency relations between it and the region higher in the containment tree. This illustrates the inherently decentralized nature of the algorithm, as it is only nodes in regions between the surrounded and surrounding region components that have all the information necessary to compute the answer.

This module works by the broadcasting of a `cont` message containing the current region component's parent ($p'$) and id ($rid'$). Nodes will only process one of these messages if their parent value is null and if the received parent id is different from the nodes region component id (lines 2–3). The parent value must be null so that nodes process only one valid `cont` message instead of indefinitely passing the message back and forth between their neighbors. The received parent id must not be the same as the node's region component id as there can be cases where the message may double back into a region that has already been partially processed. Figure 20 provides an example of this.

Nodes receiving this message with the same *rid* will set their parent value to that of the received message before rebroadcasting the message (lines 4–6), whereas nodes with a different parent value (i.e., the message has entered a new region component) will set their parent value as that of the received region component id before rebroadcasting the message (lines 7–9). The nodes will then populate their children set with any regions in their boundary table that are not the parent node before detecting any surrounds relations (lines 10–11).




a. Message injection at node 1, passed to nodes 2 and 4.

b. Message passed to nodes 3 and 5.

c. Message passed to nodes 6 and 7.

d. Node 7 has selected the incorrect parent.

Figure 20: Example showing how without a check that the received message's parent value is not equal to the node's region component id, Module 4 can cause nodes to select the incorrect parent.

To ensure that the containment tree will have the correct root, the module begins with the injection of a `cont` message to a node on the outermost component of the region with -1 set as the parent id. Using the complex areal object in Figure 16c as an example, the module begins with the injection of the message to a node in region component 1 of the form (`cont`, -1, 1). This message will allow all nodes in region 1 to select -1 as their parent (i.e., they have no parent as that region component is the region's exterior) and $\{2\}$ as their children.

Once the message reaches region 2, the parent will be set to 1, the children will be set to $\{3,4,5,6\}$, and the message will change to (`cont`, 1, 2). Using their knowledge of the containing region (*parent*) and the Voronoi-adjacency relations ($A_t$), the nodes will then compute the distance from the containing region to each of the children and therefore detect that regions 5 and 6 surround region 4.

The message will further continue on to regions 3, 4, 5, and 6, with these regions setting their parent to 2 and their children to $\{7,8\}$ for 3 and $\varnothing$ for 4, 5, and 6. Finally the message will reach regions 7 and 8 in the form (`cont`, 2, 3), with these regions setting their parent to 3 and their children to $\varnothing$.

**Module 4** Containment tree generation

1: Local variables: $parent : V \to \mathbb{N} \cup \{-1\}$, initialized to $\mathring{parent} := \varnothing$; $children : V \to 2^{\mathbb{N}}$, initialized to $\mathring{children} := \varnothing$, Surrounds table, $S_t = \langle S : (\text{set of surrounding regions}), c : (\text{surrounded region}) \rangle$, initialized with zero records.

REGN

2: *Receiving* (`cont`, $p'$, $rid'$)
3: **if** $\mathring{parent} = \varnothing$ AND $p' \neq \mathring{rid}$ **then**
4: **if** $\mathring{rid} = rid'$ **then**
5: **set** $\mathring{parent} := p'$
6: **broadcast** (`cont`, $p'$, $rid'$)
7: **else**
8: **set** $\mathring{parent} := rid'$
9: **broadcast** (`cont`, $rid'$, $\mathring{rid}$)
10: **set** $\mathring{children}$ := SELECT *bid* FROM $B_t$ WHERE $bid \neq \mathring{parent}$
11: Detect any surrounds relations and insert into $S_t$

### 4.1.7 *Module 5: Node movement*

The final module is tasked with ensuring that nodes receive the correct values when entering a different region component, specifically keeping the information obtained from modules 3 and 4 updated. Whenever a node moves to a different region (i.e., their sensed value changes) (line 1), it must perform four tasks.

Firstly, the region and Voronoi region component identifiers (*rid* and *adj*) must be corrected. This is done by swapping them (lines 2–4). Recall that the id of the Voronoi region component is also the id of the node's closest adjacent region so swapping these values will produce the desired results. Occasionally, nodes will transition to a new region component directly from a Voronoi region boundary (i.e, their *adj* value before swapping is -1), causing their new region component id to be set as -1. To correct for this, lines 15–16 were added to module 2 so that nodes with this value, on the receipt of a `hop` message from the new region with a hop count of 1, will update their region component id accordingly.

Secondly, the node's other values must be cleared. This includes the boundary, Voronoi-adjacency and surrounds tables ($B_t$, $A_t$, $S_t$) as well as the parent and children variables (lines 5–6). Thirdly, if the Voronoi region component id does not equal -1, the nodes will insert a new record into their boundary table containing the Voronoi region component id and a hop count of 1 (lines 7–8). This is done to ensure that nodes that have just transitioned region components (and therefore have cleared their boundary tables) will have an accurate record in their boundary table. Lastly, nodes will broadcast a `rqst` message along with their sensed value to request the information cleared in the second step from their neighbors (line 9).

Nodes receiving this request will respond if the received sensed value matches the node's own (i.e., the message is from a node in the same region) and if the node has a non-empty Voronoi-adjacency table (i.e., the node hasn't also just transitioned and thus cleared its variables). The node will then respond by sending a `rspc` message along with its sensed value, Voronoi-adjacency and surrounds tables as well as its parent and children variables (lines 10–12).

Nodes receiving this response will again check to see that the received sensed value matches the node's own (lines 13–14). This is necessary for cases where there is both an arrival and departure of nodes from a region component in close proximity (e.g., nodes moving from region component 1 to 2 as well as from 2 to 1) as nodes could be updated with records from the wrong region component. If the message is from the same region component and the Voronoi-adjacency table is empty, then it will be filled using records from the received message (lines 15–16). If the parent variable is set to null, then both the parent and children variables, along with the surrounds table, will be updated with corresponding records from the received message (lines 17–20).

**Module 5** Node movement

REGN

```
 1: When ŝ(now) changes
 2:     let tmp := r̊id
 3:     set r̊id := å̊dj
 4:     set å̊dj := tmp
 5:     DELETE FROM B_t, A_t, S_t
 6:     set p̊arent, ch̊ildren := ∅
 7:     if å̊dj ≠ −1 then
 8:         INSERT INTO B_t VALUES (å̊dj, 1)
 9:     broadcast (rqst, ŝ(now))
10: Receiving (rqst, s′)
11:     if ŝ(now) = s′ AND A_t ≠ ∅ then
12:         broadcast (rspc, ŝ(now), A_t, p̊arent, ch̊ildren, S_t)
13: Receiving (rspc, s′, A′_t, parent′, children′, S′_t)
14:     if ŝ(now) = s′ then
15:         if A_t = ∅ then
16:             set A_t := A′_t
17:         if p̊arent = ∅ then
18:             set p̊arent := parent′
19:             set ch̊ildren := children′
20:             set S_t := S′_t
```

The reason that the Voronoi-adjacency table updating has been separated from the updating of the other records is that there are occasionally cases where a node has entered a new region and then received and processed a `cont` message (i.e., assigned itself parent and children variables along with populating its surrounds table) but has

not yet received a `rspc` message. If it were to receive a `rspc` message from a node that has not yet processed its own `cont` message (i.e., not yet assigned itself parent and children variables or populated its surrounds table), then updating all of these values would erase its parent and children variables along with any surrounds table records.

### 4.1.8 *Algorithm summary*

With these five modules, a complete decentralized algorithm is produced that is capable of extracting high level knowledge from low level sensor data provided by individual nodes in the network. Additionally, the algorithm does this without access to the coordinate information of the nodes while accommodating node mobility.

That is not to say that the algorithm is without drawbacks. Specifically, nodes within a region component are only aware of the region component containing them and any region components they contain. This lack of a complete picture of the internal structure of the network may be adequate for some cases, such as detecting any surrounds relations. However on other occasions, it may be that the entire containment tree and Voronoi-adjacency graph is required. To build this containment tree and Voronoi-adjacency graph, information from each of the region components would need to be retrieved.

Additionally, this algorithm stores the containment tree and Voronoi-adjacency relations separately. The containment tree segments are stored as *parent* and *children* variables for each region component whereas the Voronoi-adjacency relations are stored in the Voronoi-adjacency table ($A_t$), which is also confined to a single region component.

## 4.2 SIMPLIFIED MAPTREE FOR SIMPLE REGIONS

As mentioned in the previous section, using a containment tree overlain with the Voronoi-adjacency relations is suitable for detecting surrounds at a local level. However, the drawbacks of having the information required stored in segments throughout the entire network as well as in two separate formal models is inefficient. What is required is a single formal model and corresponding data structure that is capable of efficiently storing containment and Voronoi-adjacency. For these reasons, the maptree (introduced in section 2.2) was chosen. As previously stated, the maptree is a black-white edge labeled tree based on combinatorial maps and adjacency trees capable of completely representing the topological structure of regions. Additionally, it was demonstrated in section 2.2.1 that it is possible to construct a maptree based on a DCEL table, provided that the table is first processed to split the halfedges into arbitrarily named connected components.

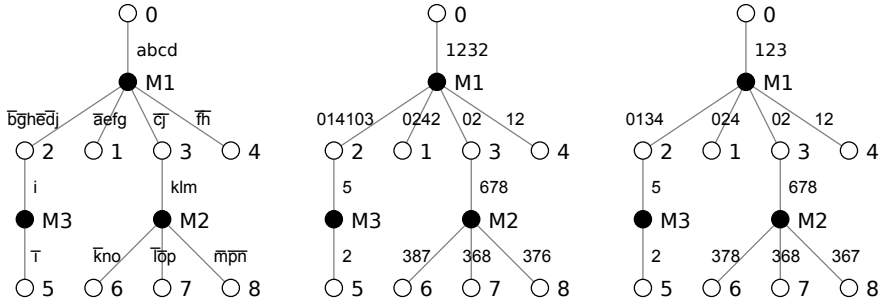



Figure 21: Maptrees: a. Standard maptree, derived from DCEL; b. Partially simplified maptree (with preserved boundary cycle); and c. Simplified maptree.

Of note in Figure 21.a is that the maptree has a defined root consisting of the region component 0. Maptrees represent the relative relationships between the regions and components, meaning that for a particular planar embedding to be specified, a root node must be selected. This root node is the unbounded exterior of the region, which in Figure 22, is region component 0. Selecting this node will allow for topological relationships to be inferred correctly. It is important to note that DCEL tables, such as the one presented in table 2, must also specify the root node. This has been done using the first record by selecting -1 for all columns other than face.

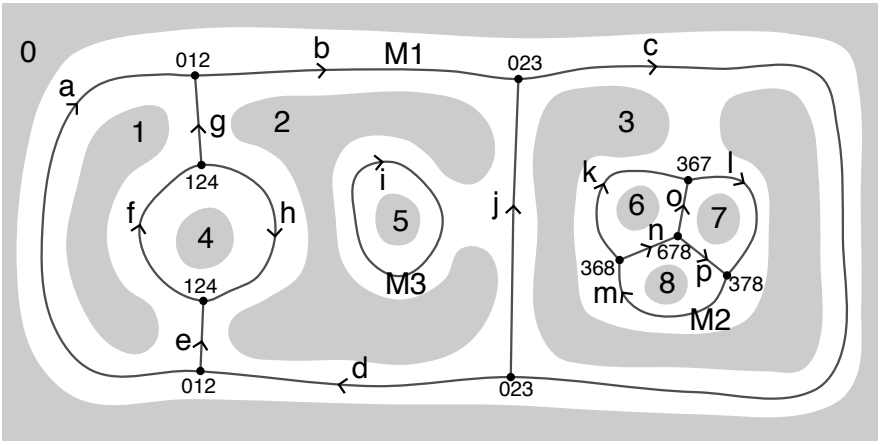



Figure 22: Simple region example with all edges labeled. Region components are grey and black lines represent the Voronoi boundaries of those region components. Black dots represent the Voronoi junctions.

For simplicity, this section will only cover simple regions, which contain only a single hole (i.e, the containment tree has a maximum diameter of three). Consider the example case of the simple static regions provided in Figure 22. It is possible to construct a maptree from the Voronoi-adjacency relations given that each edge has both

a label and an orientation, as shown by Figure 21.a, or from a DCEL table containing these edges (table 2).

| Halfedge | Twin | Next | Previous | Face | Component |
|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | 0 | -1 |
| a | $\bar{a}$ | b | d | 0 | M1 |
| $\bar{a}$ | a | e | g | 1 | M1 |
| b | $\bar{b}$ | c | a | 1 | M1 |
| $\bar{b}$ | b | $\bar{g}$ | j | 2 | M1 |
| i | $\bar{i}$ | i | i | 2 | M3 |
| $\bar{i}$ | i | $\bar{i}$ | $\bar{i}$ | 5 | M3 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Table 2: DCEL table augmented with connected component labeling. Table shows some entries for Figure 22. The completed table can be found in the appendices as Table 14.

Given that the algorithms used to detect the Voronoi boundaries only specify whether a boundary is present between two Voronoi region components, unique edge identification is not possible. For example, the edges b and d from Figure 22 would be represented by the single entry $\{0, 2\}$ in the Voronoi adjacency table $A_t$. Because these edges cannot be uniquely represented, boundary cycling of the edges (i.e., the order of edges that surround a face) is also not possible.

As unique edge identification is not possible, the formal model will need to be simplified so that only the region components adjacent to the boundary cycle of another region component are listed in the maptree edges. As each halfedge has a region its twin is adjacent to, a mapping function can be defined to replace halfedges with their associated region. This mapping is defined with the *simplify* function; i.e., *simplify*: *halfedge* $\rightarrow$ *face*. The result of applying this to the maptree in Figure 21.a. is shown in Figure 21.b. At this stage, Voronoi-adjacency boundaries are no longer uniquely represented as edges although boundary cycling is still preserved. This can be further simplified by removing duplicate entries, creating a simplified maptree as shown in Figure 21.c. At this point boundary cycling is no longer preserved.

### 4.2.1 *Simplified maptree table*

While it has been demonstrated that it is possible to construct a maptree from a DCEL table and to convert a regular maptree to a simplified one, it is also possible to simplify a DCEL table to store only information required to construct a simplified maptree. Most of this simplification comes from the fact that boundary cycling is not pre-

served (eliminating the previous and next columns) and halfedge labeling is not required (eliminating the halfedge and twin columns).

It is however not sufficient to simply store the unique entries of the face and component columns as this would only preserve the maptree edges and not their edge labels. To preserve the edge labels, a triple consisting of two faces and a connected component must be stored, indicating that these two Voronoi region components are adjacent to each other through a specific connected component. This can be done by taking the face and connected component of one half edge and pairing it with the face of its twin.

For example, in table 2 the halfedge $a$ is part of connected component M1 and faces Voronoi region component 0 whereas its twin $\bar{a}$ faces region component 1. This would create the triple $\langle 0, 1, M1 \rangle$, which can be confirmed visually by looking at Figure 22. To convert these triples into a maptree, the edge (0, M1) would be added along with a label of 1. Additionally, the inverse edge (1, M1) would be added with a label of 0. As seen in table 3, by processing all of these triples, the entire simplified maptree can be constructed.

| $rid_a$ | $rid_b$ | $cid$ |
|---|---|---|
| -1 | 0 | -1 |
| 0 | 1 | M1 |
| 0 | 2 | M1 |
| 1 | 2 | M1 |
| 2 | 5 | M3 |
| ⋮ | ⋮ | ⋮ |

Table 3: Simplified maptree table based on Table 2. The completed table can be found in the appendices as Table 15.

### 4.2.2 *Qualitative relations*

Like Figure 16 from the previous section, Figure 22 also illustrates regions on the same level of the containment tree that exist in different configurations. Again, it can be intuitively said that region component 4 is surrounded by region components 1 and 2, region component 5 is surrounded by region component 2, and that region components 6, 7, and 8 are surrounded by region component 3. Looking at the simplified maptree in Figure 21, while region component 5 and region components 6, 7 and 8 are both surrounded by a single region component and are two levels lower on the simplified maptree than their surrounding region components, region component 4 is at the same level as its surrounding region components. It is this distinction that requires the splitting of the surrounds definition into *engulfs* for when a single region component partially encloses another

region component and *surrounds* when multiple region components partially enclose another region component.

More formally, consider a simplified maptree $M$, with the set of black nodes $B$ representing the connected components of the simplified maptree and the set of white nodes $W$ representing the set of Voronoi region components. For a particular Voronoi region component $x \in B \cup W$, $level(x)$ is written to refer to its level in the simplified maptree; i.e., $level$: $B \cup W \rightarrow \mathbb{N}$.

Because the Voronoi region components are induced by region components, there is a 1:1 mapping between them. This is defined with the $gen$ function. In other words, $gen(w)$ or $gen(W)$ are the regions that generate white nodes $w$ or $W$. Voronoi regions that share a connected component with a specified region component are then determined using the *abut* function. These are the region components that are Voronoi-adjacent to that region component with respect to the specified boundary and are written on the edges of the simplified maptree; i.e., $abut$: $B \times W \rightarrow 2^W$. With this, the surrounds and engulfs relations can be defined as follows:

**Definition** Consider Voronoi region components $w_1, w_2 \in W$ such that $level(w_1) < level(w_2)$ and there exists a $b \in B$ where $bw_1, bw_2 \in M$. If $w_1 \notin abut(b, w_2)$ then $gen(abut(b, w_2))$ is said to *surround* $gen(w_2)$

**Definition** Let $w_1, w_2 \in W$ such that $level(w_1) < level(w_2)$ and there exists a $b \in B$ where $bw_1, bw_2 \in M$. Then $gen(w_1)$ *engulfs* $gen(w_2)$

Informally, a region component $c$ is surrounded by regions $S$ if $c$ is Voronoi-adjacent to all of $S$, and both $c$ and all of $S$ are at the same level in the simplified maptree. Additionally, a region component $c$ is engulfed by region $s$ if $s$ is exactly two levels higher in the simplified maptree. With this, it is now possible to both store the simplified maptree as well as calculate the surrounds and engulfs relations for simple regions.

### 4.2.3 *Algorithm design*

In implementing this new formal model, some aspects of the original algorithm must be changed. While the original algorithm is capable of handling complex region configurations (i.e., multiple nested positive and negative region components), for simplicity's sake the algorithm of this section will only be capable of functioning on simple region configurations. As such the algorithm will in practice only ever detect a single negative region component. In the new algorithm, modules 1 and 2 will remain unchanged, with the algorithm still selecting and distributing unique ids for each (positive or negative) region component in the network and determining the approximate Voronoi boundaries between region components. This algorithm will be referred to

as the simpleStatic algorithm, and the relations between the modified modules can be seen in Figure 23.

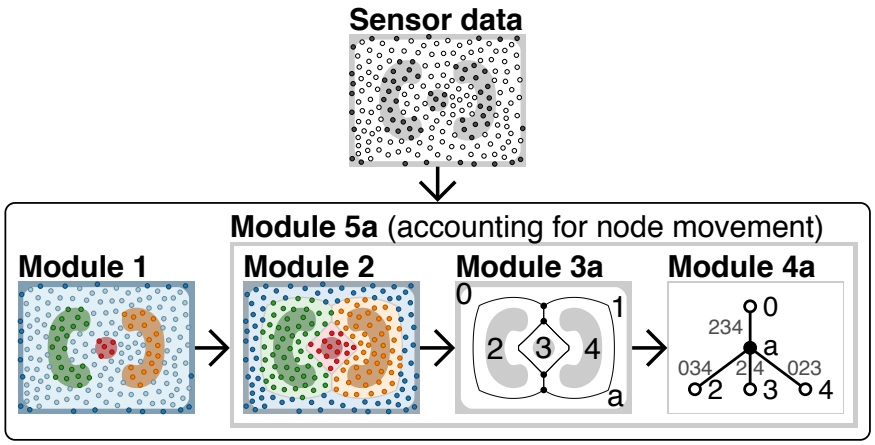



Figure 23: Flow diagram representing the interactions between the modules that comprise the simpleStatic algorithm.

Broadly speaking, the new modules must be able to do the following three things in order to make use of the simplified maptree:

1. Detect Voronoi-adjacency relations,
2. Break them up into connected components, and
3. Store the simplified maptree components in a table of the form (*region component id a*, *region component id b*, *connected component id*).

Using Figure 22 as an example, the original module 3 is already capable of storing the Voronoi-adjacency relations in its Voronoi-adjacency table ($A_t$). To break these sets of relations into a set of connected components, Voronoi-adjacency junctions will also have to be stored. These junctions are points where three Voronoi region components directly abut each other and can be detected by minor modifications to the module.

Using the Voronoi-adjacency junctions, module 4 will split the regions into connected components (shown as black lines in Figure 22). This is done by grouping junctions that share two regions with at least one other member in the group. For example, $(0,1,2)$, $(0,2,3)$, and $(1,2,4)$ are all connected as they share two values between at least one other member. This gives us the connected components $\{0,1,2,3,4\}$ and $\{3,6,7,8\}$, which have been given the arbitrary names M1 and M2.

From the diagram, it can be seen that there are actually 3 connected components, with the middle one not being registered. This is because it has no Voronoi-adjacency junction associated with it. To correct for this, a search of the Voronoi-adjacency pairs must be made for

pairs where one of the regions is not present in any of the connected components. In this case, it would be the pair 25, which can be added as M3: $\{2, 5\}$.

To store the simplified maptree records, all that is needed is to add an additional column to the Voronoi-adjacency table, populating it with the id of the associated connected component. For adjacency relation (0,1) this would involve finding which connected component contains both region component ids. In this case, that would be M1, creating the entry (0,1,M1).

The detection of the surrounds and engulfs relations both rely on a level function, which returns the distance a specified node is from the root of the simplified maptree. For this to work, the id of the root region component (i.e, the exterior of the region) must be known. This can be done by either having a node on the outermost region component of the region configuration use surprise flooding to distribute the root node id throughout the network, or for the network to simply be initialized knowing the root (i.e., the exterior region component has a specific id).

To explain the specifics of implementing this model, the following sections, which include a description of the modified modules and their pseudocode, have been provided.

### 4.2.4 *Module 3a: Modified adjacency relation identification*

Like the original module, surprise flooding is used to propagate the Voronoi-adjacency relations to all nodes within a region component, although this information is now stored within the simplified maptree table ($M_t$) instead of the Voronoi-adjacency table ($A_t$). Unlike the Voronoi-adjacency table, the simplified maptree table has a third column for storing the id of the connected component that relation is a part of. As this id is assigned by the next module, this column can be ignored here. This module has been additionally modified to detect and propagate Voronoi-junctions, which are points where three Voronoi region components directly abut each other.

As in the original module, nodes on a boundary between adjacent Voronoi regions (i.e, nodes with an *adj* value of -1) will create a set of the region components that make up this boundary (lines 2–5). Nodes on a boundary between two Voronoi region components will add this record to their simplified maptree table ($M_t$) and broadcast boundary and junction request messages (`bndy` and `junq`) to their neighbors (lines 6–9).

Nodes on the boundary between three Voronoi region components will insert these values into their Voronoi-junction table ($J_t$) and broadcast a junction found message (`junf`) containing this adjacency triple along with their sensed value to their neighbors (lines 10–12).

Nodes receiving a junction request message (junq) will check to see if they have the same sensed value (i.e., within the same region), and that their *adj* value is not a member of the received pair and is not -1 (i.e., the node is not on a Voronoi region component boundary). Nodes capable of meeting these requirements would be nodes in one Voronoi region component with neighbors on the boundary of two other Voronoi region components. Using Figure 24, the Voronoi-adjacency junction would be $\{1, 2, 3\}$ (lines 17–18).

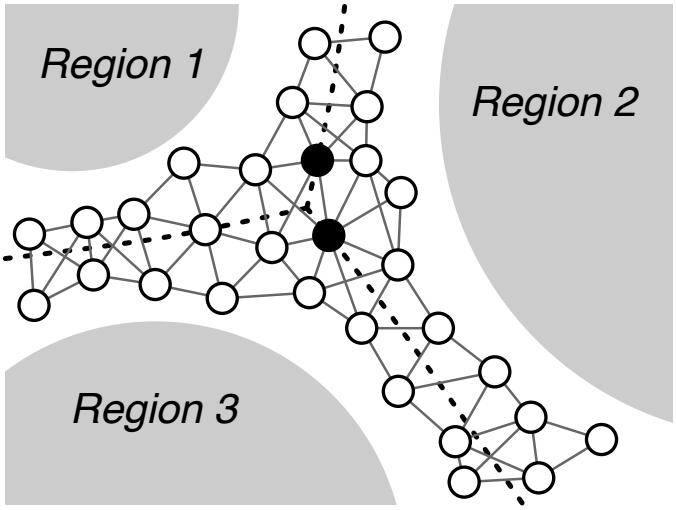



Figure 24: Adjacency example where no node is on the boundary between the three Voronoi region components. Nodes capable of detecting the junction $\{1, 2, 3\}$ by communicating with their neighbors are shaded black.

These nodes will then aggregate the ids of these three Voronoi region components into a junction set before broadcasting it to their neighbors in a junction found message along with their sensed value. The junction will then be added to the node's Voronoi-junction table if it is not already present (lines 19–22).

Nodes receiving a boundary (bndy) or junction found (junf) message will check to see that they haven't stored the received entry yet. If this is the case, they will add it to the corresponding table before rebroadcasting the message (lines 13-16, 23–26). Unlike the original module, these messages require no sensed value check, meaning that on completion of this module, all records for the simplified maptree table and Voronoi-junction table will be stored in each node of the network.

**Module 3a** Modified adjacency graph propagation

1: Local variables: Simplified maptree table, $M_t = \langle rid_a : \mathbb{N}, rid_b : \mathbb{N}, cid : \mathbb{N} \rangle$, initialized with record $(-1, root, -1)$ where *root* is the id of the region component comprising the exterior of the region, Voronoi-junction table $J_t = \langle rid_a : \mathbb{N}, rid_b : \mathbb{N}, rid_c : \mathbb{N} \rangle$, initialized with zero records.

REGN

2: *When* adjacency timer elapsed
3: **if** $\mathring{adj} = -1$ **then**
4: **let** *MinHop* := SELECT MIN $(h)$ FROM $B_t$
5: **let** *closest* := SELECT *bid* FROM $B_t$ WHERE $h = MinHop$
6: **if** $|closest| = 2$ **then**
7: INSERT INTO $M_t$ VALUES *closest*
8: **broadcast** (`bndy`, *closest*)
9: **broadcast** (`junq`, $\mathring{s}(now)$, *closest*)
10: **if** $|closest| = 3$ **then**
11: INSERT INTO $J_t$ VALUES *closest*
12: **broadcast** (`junf`, *closest*)
13: *Receiving* (`bndy`, *pair*)
14: **if** $pair \notin M_t$ **then**
15: INSERT INTO $M_t$ VALUES *pair*
16: **broadcast** (`bndy`, *pair*)
17: *Receiving* (`junq`, $s'$, *pair*)
18: **if** $\mathring{s}(now) = s'$ AND $adj \notin pair$ AND $adj \neq -1$ **then**
19: **let** $junction := pair \cup adj$
20: **if** $junction \notin J_t$ **then**
21: **broadcast** (`junf`, *junction*)
22: INSERT INTO $J_t$ VALUES *junction*
23: *Receiving* (`junf`, $J_t$)
24: **if** $junction \notin J_t$ **then**
25: INSERT INTO $J_t$ VALUES *junction*
26: **broadcast** (`junf`, *junction*)

### 4.2.5 *Module 4a: Simplified maptree generation*

Now that each node has the complete set of Voronoi-adjacency relations and Voronoi-adjacency junctions stored in their respective tables, once the maptree timer elapses, module 4 will assign connected components to each entry in the simplified maptree table. This timer is set to elapse at a set period after the adjacency timer so that every node has the complete set of information necessary to construct the simplified maptree.

First, each entry in the Voronoi-junction table ($J_t$) is added as a set to the collection *components* (Line 3). Each set is then compared with every other set to see if they share at least two region components in common (Lines 4–6). If this is the case, these sets are merged together (Line 7). Once completed, the example region from figure 22 would produce the following collection of connected components: $\{\{0, 1, 2, 3, 4\}, \{3, 6, 7, 8\}\}$.

However, some connected components have no junctions within themselves, meaning that they are comprised of a single Voronoi-adjacency pair. To correct for this, all region components present within the simplified maptree table that are not already present in the *components* collection are added to the *remaining* set (Line 8). Records from the simplified maptree table that match one of these region components are then added to the *components* collection (Lines 9–10). This would make the example collection of connected components: $\{\{0,1,2,3,4\},\{3,6,7,8\},\{2,5\}\}$.

Lastly, every entry of the simplified maptree table is sorted through to find each region's associated connected component (Lines 11–12). This will be the connected component that contains both region component ids from the simplified maptree table's entry (Line 13). This record's *cid* field will then be updated with the label of the connected component (Line 14). This label is obtained using the *clabel* function, which determines a unique id for a set of region component ids.

Once the simplified maptree table has been completed, it can, along with the root node, be used to determine any surrounds or engulfs relations.

**Module 4a** Simplified maptree generation

1: Restrictions: component labeling function: $clabel(c) \rightarrow \mathbb{N}$ where $\mathbb{N}$ is a unique id for that set of region components.

REGN
2: *When* maptree timer elapsed
3: **let** *components* := *collection of sets derived from entries from* $J_t$
4: **for all** $i \in components$ **do**
5: **for all** $j \in components$ **do**
6: **if** $|i \cap j| \geq 2$ **then**
7: MERGE $(i, j)$
8: **let** *remaining* := *set of all unique region ids from* $M_t$ *not present in components*
9: **for all** $k \in remaining$ **do**
10: **let** $components := components \cup \{$SELECT $rid_a, rid_b$ FROM $M_t$ WHERE $rid_a = k$ OR $rid_b = k\}$
11: **for all** $l \in M_t$ **do**
12: **for all** $m \in components$ **do**
13: **if** $\{l\} \subseteq m$ **then**
14: UPDATE $l$ SET $cid = clabel(m)$

### 4.2.6 *Module 5a: Modified node movement*

As the algorithm only has to contend with simple regions, this module has been greatly simplified from its original form. As each node has the same entries for the simplified maptree table and Voronoi-junction table, these do not need to be refreshed when a node enters a new region. Because of this, only the boundary table ($B_t$) needs to

be refreshed (Lines 5–7), and the region component and Voronoi region component id's need to be updated, which is done by swapping them (Lines 1–4).

**Module 5a** Modified node movement

REGN
1: *When* $\mathring{s}(now)$ *changes*
2: **let** $tmp := \mathring{rid}$
3: **set** $\mathring{rid} := \mathring{adj}$
4: **set** $\mathring{adj} := tmp$
5: DELETE FROM $B_t$
6: **if** $\mathring{adj} \neq -1$ **then**
7: INSERT INTO $B_t$ VALUES $(\mathring{adj}, 1)$

## 4.3 SIMPLIFIED MAPTREE FOR COMPLEX REGIONS

While the algorithms of the previous section are sufficient for determining the simplified maptree for simple regions, additional modifications will be required to accurately describe the simplified maptrees in complex areal objects, an example of which is shown in Figure 25. In contrast to the simple region example of Figure 22, this complex region example has two sets of Voronoi-adjacency boundaries, induced by the positive and the negative region components respectively.

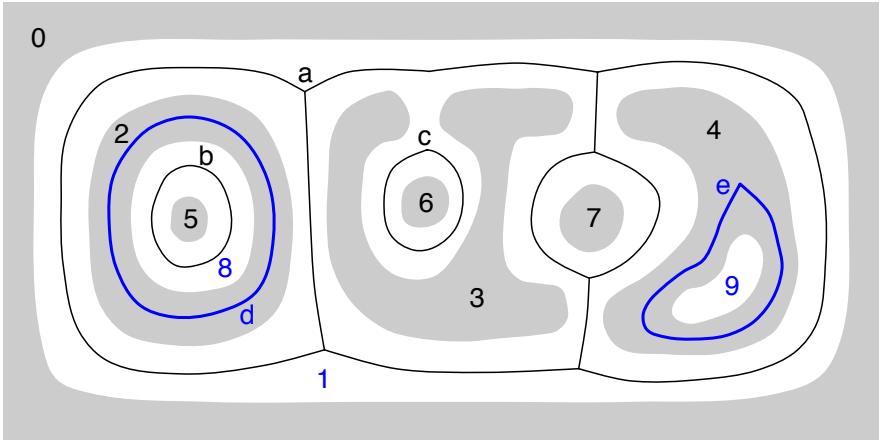



Figure 25: Complex regions example. Positive region components are grey and negative region components are white. Black lines represent the Voronoi boundaries induced by positive region components and blue, thick lines represent the Voronoi boundaries induced by negative region components.

### 4.3.1 *Qualitative relations*

Like the simple region example, these two sets of Voronoi-adjacency boundaries can be collected into simplified maptrees, as shown in Fig-

ure 26. Formally, $M^+$ and $M^-$ will refer to the simplified maptrees of the positive and negative region components respectively. Additionally, $B^+, W^+, B^-$, and $W^-$ will refer to the sets of black and white nodes for the simplified maptrees of the positive and negative region components. These can be aggregated to represent the entire set of black and white nodes; i.e., $B = B^+ \cup B^-$ and $W = W^+ \cup W^-$.

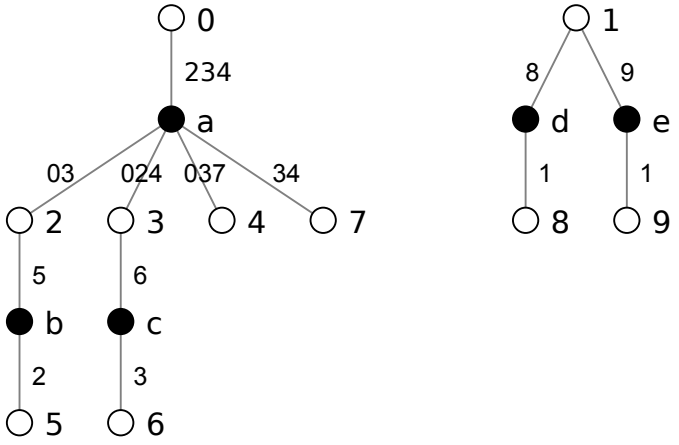



Figure 26: Simplified maptree example based on complex region from Figure 25. Simplified maptree induced by the positive region components ($M^+$) on left and simplified maptree induced by the negative region components ($M^-$) on right. The simplified maptree table can be found in table 16 of the appendix.

Like the simple region example, the complex region example illustrates both the surrounds and engulfs relations. In both the region configuration (Figure 25) and the positive simplified maptree (Figure 26), the positive region component 7 is surrounded by region components 3 and 4 whereas region component 6 is engulfed by region component 3. Judging by the positive simplified maptree it appears that region component 5 is engulfed by region 2. However, looking at the region configuration in Figure 25, it is clear that region component 5 is in fact *contained* by region 2. Writing this as a formal definition for detecting engulf and contain relations in the positive simplified maptree is as follows:

**Definition** Let $w_1, w_2 \in W^+$ such that $level(w_1) < level(w_2)$ and there exists a $b \in B^+$ where $bw_1, bw_2 \in M^+$. Then $gen(w_1)$ *engulfs* or *contains* $gen(w_2)$

In order to distinguish between engulf and contain relations, information is additionally required from the opposite simplified maptree. Specifically, each connected component is contained within the Voronoi region component of its opposite simplified maptree. This relationship is defined by the *label* function, i.e., $B^+ \rightarrow W^-$. An example output of this function for the object in Figure 25 is presented in Table 4.

| Input | Output |
|---|---|
| a | 1 |
| b | 8 |
| c | 1 |
| d | 2 |
| e | 4 |

Table 4: Table of results of *label* function for connected components of Figure 25.

From this table it can be seen that connected components $a$ and $c$ are within the same region component and that connected components $a$ and $b$ are not. It is this difference that allows for a formal definition of the distinction between engulfs and contains:

**Definition** Consider $w_1, w_2 \in W^+$ where $gen(w_1)$ *engulfs* or *contains* $gen(w_2)$. If there exists $b_1, b_2 \in B^+$ where $b_1w_1, w_1b_2, b_2w_2 \in M^+$ and $label(b_1) = label(b_2)$ then $gen(w_1)$ *engulfs* $gen(w_2)$. Otherwise $gen(w_1)$ *contains* $gen(w_2)$.

With this, it is now possible to store the simplified maptree for both positive and negative region components as well as calculate surrounds, engulfs and contains relations for complex regions.

### 4.3.2 *Algorithm design*

To extend the algorithm designed in the previous section so that it is capable of working with complex regions, some key changes must be made. Specifically, a table must be added similar to Table 4 to store the region components the connected components reside within. This is for the *label* function to be able to distinguish between engulfs and contains relations. Additionally, a new column must be added to the simplified maptree table ($M_t$) in order to distinguish entries that are part of the positive and negative simplified maptrees. This algorithm will be referred to as the complexStatic algorithm, and the relations between the modified modules are identical to those shown for the simpleStatic algorithm in Figure 23.

To explain the specifics of implementing this model, the following sections, which include a description of the modified modules and their pseudocode, have been provided.

### 4.3.3 *Module 3b: Modified adjacency relation identification for complex regions*

Module 3 runs in essentially the same way as its counterpart from the previous section with the exception that the simplified maptree

and Voronoi-Junction tables ($M_t$ and $J_t$) are restricted to the region components that calculated them. Additionally, the simplified maptree table has a *neg* column to indicate whether the entry is part of the positive maptree or negative maptree, with 1 indicating that the entry is part of the negative simplified maptree and 0 indicating that the entry is part of the positive simplified maptree.

**Module 3b** Modified adjacency graph propagation for complex regions

1: Local variables: Simplified maptree table, $M_t = \langle rid_a : \mathbb{N},\ rid_b : \mathbb{N},\ cid : \mathbb{N},\ neg : \{0,1\}\rangle$, initialized with record $(-1, root, -1, 0)$ where *root* is the id of the region component comprising the exterior of the region, Voronoi-junction table $J_t = \langle rid_a : \mathbb{N},\ rid_b : \mathbb{N},\ rid_c : \mathbb{N}\rangle$, initialized with zero records.

REGN
2: *When* adjacency timer elapsed
3: **if** $\mathring{adj} = -1$ **then**
4: **let** *MinHop* := SELECT MIN $(h)$ FROM $B_t$
5: **let** *closest* := SELECT *bid* FROM $B_t$ WHERE $h = MinHop$
6: **if** $|closest| = 2$ **then**
7: INSERT INTO $M_t$ VALUES *closest*
8: **broadcast** (`bndy`, $\mathring{s}(now)$, *closest*)
9: **broadcast** (`junq`, $\mathring{s}(now)$, *closest*)
10: **if** $|closest| = 3$ **then**
11: INSERT INTO $J_t$ VALUES *closest*
12: **broadcast** (`junf`, $\mathring{s}(now)$, *closest*)
13: *Receiving* (`bndy`, $s'$, *pair*)
14: **if** $\mathring{s}(now) = s'$ AND $pair \notin M_t$ **then**
15: INSERT INTO $M_t$ VALUES *closest*
16: **broadcast** (`bndy`, $\mathring{s}(now)$, *pair*)
17: *Receiving* (`junq`, $s'$, *pair*)
18: **if** $\mathring{s}(now) = s'$ AND $adj \notin pair$ AND $adj \neq -1$ **then**
19: **let** *junction* := $pair \cup adj$
20: **if** $junction \notin J_t$ **then**
21: **broadcast** (`junf`, $\mathring{s}(now)$, *junction*)
22: INSERT INTO $J_t$ VALUES *junction*
23: *Receiving* (`junf`, $s'$, $J_t$)
24: **if** $\mathring{s}(now) = s'$ AND $junction \notin J_t$ **then**
25: INSERT INTO $J_t$ VALUES *junction*
26: **broadcast** (`junf`, $\mathring{s}(now)$, *junction*)

### 4.3.4 *Module 4b: Simplified maptree generation for complex regions*

Now that each node has the set of Voronoi-adjacency relations and Voronoi-adjacency junctions detected by the region component they are presently in stored in their respective tables, module 4 will once again assign connected components to each entry in the simplified maptree table. This works almost identically to the module from the previous section; where this module differs is that once this is com-

plete, the simplified maptree table then fills its *neg* column with its sensed value (Line 16). Recall that a sensed value of 0 or 1 indicates that the node is within a negative or positive region component respectively. Directly adding the sensed value works as nodes in negative region components detect segments of positive simplified maptrees ($M^+$), and nodes in positive region components detect segments of negative simplified maptrees ($M^-$).

In order to distinguish between engulf and contain relations, the *label* function is required. To implement this, the label table ($L_t$) was added to store the ids of the connected components and the ids of region components they reside within. As all the connected components presently stored within a node have originated from that node's current region component, this table can be populated with these records and the region component id, *rid* (Lines 17–18).

**Module 4b** Simplified maptree generation for complex regions

1: Local variables: label table $L_t = \langle cid : \mathbb{N}, rid : \mathbb{N} \rangle$, initialized with zero records.
2: Restrictions: component labeling function: $clabel(c) \to \mathbb{N}$ where $\mathbb{N}$ is a unique id for that set of region components.

REGN
3: *When* maptree timer elapsed
4: **let** *components* := *collection of sets derived from entries from* $J_t$
5: **for all** $i \in components$ **do**
6: **for all** $j \in components$ **do**
7: **if** $|i \cap j| \geq 2$ **then**
8: MERGE $(i, j)$
9: **let** *remaining* := *set of all unique region ids from* $M_t$ *not present in components*
10: **for all** $k \in remaining$ **do**
11: **let** $components := components \cup \{$SELECT $rid_a, rid_b$ FROM $M_t$ WHERE $rid_a = k$ OR $rid_b = k\}$
12: **for all** $l \in M_t$ **do**
13: **for all** $m \in components$ **do**
14: **if** $\{l\} \subseteq m$ **then**
15: UPDATE $l$ SET $cid = clabel(m)$
16: UPDATE $M_t$ SET $neg = \mathring{s}(now)$ WHERE $neg = \varnothing$
17: **for all** $n \in components$ **do**
18: INSERT INTO $L_t$ VALUES $\{clabel(n), \mathring{rid}\}$
19: **broadcast** (`mapt`, $M_t$, $L_t$)
20: *Receiving* (`mapt`, $M'_t$, $L'_t$)
21: **if** $M'_t \not\subseteq M_t$ **then**
22: INSERT INTO $M_t$ VALUES $M'_t$
23: INSERT INTO $L_t$ VALUES $L'_t$
24: **broadcast** (`mapt`, $M'_t$, $L'_t$)

As each node is restricted to entries in the simplified maptree and label table, which have originated from that node's current region component, surprise flooding will be used to propagate these entries

throughout the network (Lines 19–24). This will ensure that upon the completion of this module, every node will have access to the information necessary to determine the presence of any surrounds, contains or engulfs relations.

### 4.3.5 *Module 5b: Modified node movement for complex regions*

As the algorithm only now has to account for complex regions, this module has been expanded on. Specifically, when a node changes region components, if module 4 has not yet run (i.e., the label table has no records), then the simplified maptree table and junction table will have to be cleared and replaced with new records requested from the node's new neighbors (lines 8–10).

Nodes receiving this request will respond if the received sensed value matches the node's own (i.e., the message is from a node in the same region), and if the node has a non-empty simplified maptree table (i.e., the node hasn't also just transitioned and thus cleared its variables). The node will then respond by sending a response message along with its sensed value, simplified maptree table, and Voronoi-junction table (lines 11-13).

**Module 5b** Modified node movement

```
REGN
 1: When s̊(now) changes
 2:     let tmp := ṙid
 3:     set ṙid := ȧdj
 4:     set ȧdj := tmp
 5:     DELETE FROM B_t
 6:     if ȧdj ≠ −1 then
 7:         INSERT INTO B_t VALUES (ȧdj, 1)
 8:     if L_t = ∅ then
 9:         DELETE FROM M_t, J_t
10:         broadcast (rqst, s̊(now))
11: Receiving (rqst, s′)
12:     if s̊(now) = s′ AND |M_t| > 1 then
13:         broadcast (rspc, s̊(now), M_t, J_t)
14: Receiving (rspc, s′, M′_t, J′_t)
15:     if s̊(now) = s′ AND M_t = ∅ then
16:         set M_t := M′_t
17:         set J_t := J′_t
```

Nodes receiving this response will again check to see that the received sensed value matches the node's own. This is necessary for cases where there is both an arrival and departure of nodes from a region component in close proximity (e.g., nodes moving from region component 1 to 2 as well as from 2 to 1) as nodes could be updated with records from the wrong region component. If the message is from the same region component and the simplified maptree table is

empty, then it will be filled using records from the received message (lines 14–18).

## 4.4 SUMMARY

This chapter presents three decentralized algorithms that are capable of determining the internal structure of static regions and discovering any qualitative relations that may be present. This is all done without reference to location and while the nodes making up this network are mobile. While these algorithms are all capable of extracting high level knowledge from low level sensor data provided by individual nodes in the network, there are some key differences between them:

- The first algorithm is capable of running on complex regions and describes the calculation of both the containment relationships between region components and the adjacency relations between Voronoi regions of those components. By combining these Voronoi adjacency relations with the containment tree, this algorithm is capable of detecting the presence of any surrounds relations, where one or more regions partially enclose another.

- The second algorithm is only capable of running on simple regions and introduces the simplified maptree for the storage of the adjacency relations of the Voronoi regions induced by the region components. This algorithm splits the surrounds relation into engulfs, for when a single region partially encloses another region, and surrounds, when multiple regions partially enclose another region.

- The third algorithm extends the second so that it may additionally run on complex regions. It does so by splitting the simplified maptree into two maptrees; the positive simplified maptree for relations induced by the positive region components and the negative simplified maptree for the relations induced by the negative region components. In addition to being able to detect surrounds and engulfs relations, by combining information from the two simplified maptrees, this algorithm is able to further distinguish between engulfs and contains relations.

Chapter 6 will then evaluate these algorithms in terms of scalability and veracity.

# 5 DYNAMIC REGIONS

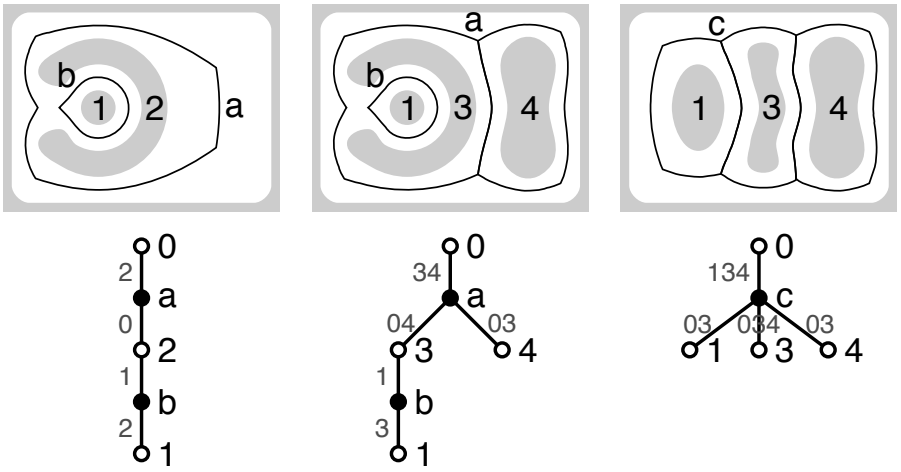
While the previous chapter presented algorithms capable of determining the internal structure of static regions and discovering any qualitative relations present, these algorithms all had the limitation that the underlying regions be static. This chapter presents two sets of decentralized algorithms that are capable of additionally handling dynamic regions. The first algorithm extends the algorithm of section 4.2, which deals with simple regions, and the second algorithm extends the algorithm of section 4.3, which deals with complex regions.

As dynamic regions are now being considered, it is important to first define the specifics of what changes are possible by the region. While this has been discussed in more detail in section 2.1.3, the key points will be restated here. For simple region configurations, the topological events of components splitting, merging, appearing, and disappearing are all possible. For the sake of simplicity, region components can split into at most two components and a maximum of two region components can merge. Additionally, these topological events will be treated as atomic, with at most one topological event occurring during a single time step. By extending this to complex regions, the events self-split and self-merge are additionally possible.

## 5.1 SIMPLE REGIONS

As the modifications necessary to extend the algorithms from the previous chapter to work with dynamic region configurations are extensive, simple regions will be considered to start with. As mentioned in section 4.2, simple regions contain only a single hole (i.e, the containment tree has a maximum diameter of three). The following section will further expand upon these extensions, allowing for complex regions to be monitored. To extend the algorithm from section 4.2, the new modules must be able to do the following three things in order to handle dynamic regions;

1. Expand the static data structure to a dynamic data structure capable of storing topological changes,
2. Detect topological changes, and
3. Correctly record these changes in the dynamic data structure.




a. t=10, region component 2 engulfs 1.

b. t=20, region component 2 has split into region components 3 and 4.

c. t=30, region component 3 no longer engulfs 1. Connected components a and b have merged into c.

Figure 27: Dynamic region example with accompanying simplified maptree showing three time steps for a simple region.

### 5.1.1 *Dynamic data structure*

Consider the example dynamic region configuration shown in figure 27. Between time periods 10 and 20 the region component 2 splits into the region components 3 and 4, and between time periods 20 and 30 region component 1 is no longer engulfed by region component 3. This removal of engulfment has also caused the region components a and b to merge into c. To store these changes in the structure of the region, the data structure must be expanded to account for when changes to connected components and region components take place. This is done by adding the columns $t_1$ and $t_2$ to the simplified maptree table, where $t_1$ records when the region or connected component is first detected and $t_2$ records when it is no longer present. An example of this is shown for the region in Figure 27 in table 5.

While this simplified maptree table is now capable of showing the start and end times of region and connected components, it makes no distinction between appearance and split events, or between disappearance and merge events. For example, at time period 20, it appears that all records of region component 2 have ended, instead replaced with similar records of region components 3 and 4. This either means that region component 2 has disappeared and region components 3 and 4 have appeared, or that region component 2 has split into the region components 3 and 4.

To aid in distinguishing this, the change table, $C_t$, is introduced to record split and merge events. Records where a region or connected

| $rid_a$ | $rid_b$ | $cid$ | $t_1$ | $t_2$ |
|---|---|---|---|---|
| -1 | 0 | -1 | 0 | ∅ |
| 0 | 2 | a | 0 | 20 |
| 1 | 2 | b | 0 | 20 |
| 0 | 3 | a | 20 | 30 |
| 0 | 4 | a | 20 | 30 |
| 1 | 3 | b | 20 | 30 |
| 3 | 4 | a | 20 | 30 |
| 0 | 1 | c | 30 | ∅ |
| 0 | 3 | c | 30 | ∅ |
| 0 | 4 | c | 30 | ∅ |
| 1 | 3 | c | 30 | ∅ |
| 3 | 4 | c | 30 | ∅ |

Table 5: Simplified maptree table ($M_t$) augmented with start and end times based on Figure 27.

component appear or disappear are then assumed to be appearance or disappearance events respectively unless they have an associated record in their change table. Table 6 shows an example of this for the region in Figure 27. The first column, $w$, shows the id of the wholly integrated region component whereas the columns $p_1$ and $p_2$ show the individual parts. The next column, $split$, then indicates if the event occurring is a split event or a merge event, with the final column, $t$, indicating when the event occurred. From the example table, it can be seen that region component 2 split into region components 3 and 4 at time period 20 and that the connected components a and b merged into connected component c at time period 30.

| $w$ | $p_1$ | $p_2$ | $split$ | $t$ |
|---|---|---|---|---|
| 2 | 3 | 4 | TRUE | 20 |
| c | a | b | FALSE | 30 |

Table 6: Change table ($C_t$) for logging split and merge events based on Figure 27.

### 5.1.2 *Qualitative relations*

By extending the data structure to account for changes to the simplified maptree, the qualitative relation detection can be extended to cover the ways in which region components both enter and leave these configurations. Formally, consider a dynamic simplified maptree $M$ where the configuration at a certain timestep $t$ is expressed as

$M_t|M_t \in M$. For a region component to enter and leave a qualitative relation, three time periods are required; where the region component is not yet in the relation ($t = i$), has entered the relation ($t = j$), and has left the relation ($t = k$); i.e., $\{M_i, M_j, M_k\} \subseteq M|i < j < k$.

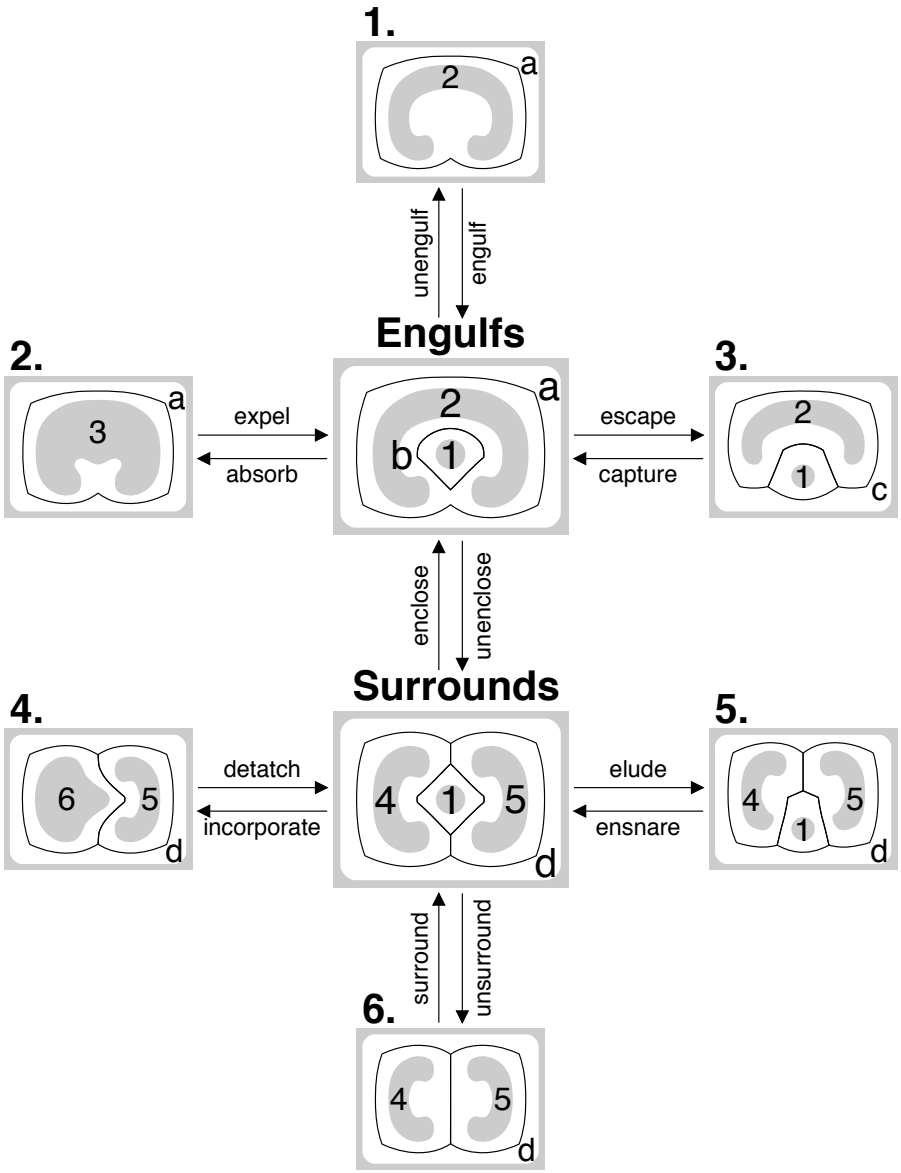



Figure 28: Conceptual neighborhood graph for simple regions showing the four possible methods region components can configure into or out of an engulfs or surrounds relation.

Looking at the conceptual neighborhood graph [23] for simple regions shown in Figure 28, there are four ways for each to transition to or from engulfs and surrounds relations. This gives a total of 24 distinct ways to enter or exit an engulfs or surrounds relation. These transitions can be split into three categories; those where a region component appears or disappears, those where a region component

splits or merges, and those where there is no topological change but a change in the internal configuration of the region. For simplicity's sake, only transitions entering the engulfs or surrounds relation will be discussed, as detecting the reverse of these transitions can be done by reversing the order of detected events.

There are two cases where region component appearance creates a qualitative relation; the transition between region configuration 1 and the engulfs relation (engulf), and the transition between region configuration 6 and the surrounds relation (surround). In both cases a new region has appeared, although in the case of engulf, a new connected component (b) has also appeared. These transitions can be found by checking for the presence of a new surrounds or engulfs relation after the appearance of a new region component.

For split events, there are three cases where a qualitative relation is created; the transition between region configuration 2 and the engulfs relation (expel), the transition between engulfs and surrounds (unenclose), and the transition between region configuration 4 and the surrounds relation (detach). In the cases of expel and detach, the region component has split to form the engulfed/surrounded region component and the engulfing or one of the surrounding region components. This is in contrast to the unenclose transition where it is the engulfing region component that has split into surrounding components. Further distinguishing these three transitions is the appearance of a connected component (b) for the expel transition and the merging of connected components (a and b merge to c) for the unenclose transitions. While the expel and detach transitions can be found by checking for the presence of a new surrounds or engulfs relation after a split event, the unenclose transition must additionally be accompanied by the merging of connected components.

There are two cases where there is a transition between qualitative relations where no topological event (i.e., appearance, disappearance, merging, and splitting) occurs; the transition between region configuration 3 and the engulfs relation (capture), and the transition between region configuration 5 and the surrounds relation (ensnare). The distinction between these two transitions is that capture causes a split of the connected component (c splits into a and b). As there is no associated topological event, these transitions can only be detected by finding engulfs or surrounds relations where there are no such events.

### 5.1.3 *Algorithm design*

To implement the dynamic simplified maptree, some aspects of the original algorithm must be changed. Like the algorithm of section 4.2, modules 1 and 2 will remain unchanged, with the modifications occurring to modules 3–5. Module 3 will have the simplified maptree

table replaced with the dynamic simplified maptree table and the change table. Additionally, module 3 will need to be rerun to detect changes in the simplified maptree.

Module 4 will group the present detected adjacency components into connected components based on the previous time period's connected component members. Finally, module 5 will have to detect the topological changes appear, disappear, merge, and split, and then rerun modules 3 and 4 periodically. This algorithm will be referred to as the simpleDynamic algorithm, and the relations between the modified modules can be seen in Figure 29. To explain the specifics of implementing this model, the following sections, which include a description of the modified modules and their pseudocode, have been provided.

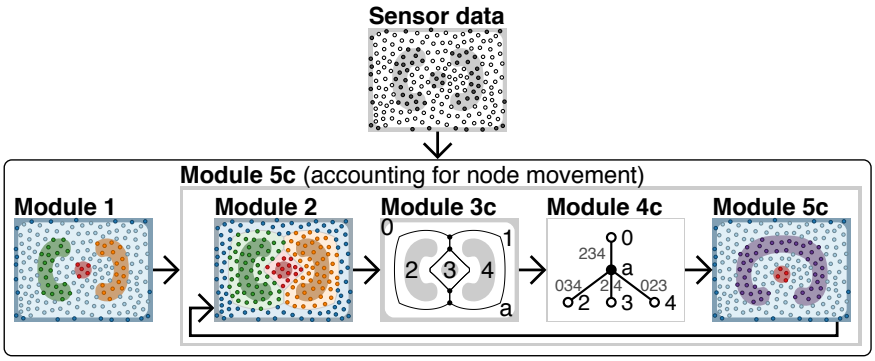



Figure 29: Flow diagram representing the interactions between the modules that comprise the simpleDynamic algorithm.

### 5.1.4 *Module 3c: Adjacency relation identification for dynamic regions*

Module 3 essentially runs in an identical way to the module from section 4.2. The main difference are that the dynamic simplified maptree table, $M_t$, has an additional two columns to represent the start and end times of a relation, and that a change table, $C_t$, has been added to track merge and split events in both region and connected components.

To ensure that changes to the configuration of the connected components are detected correctly, the junction table ($J_t$) is cleared every time this module is run (Line 4). Additionally, new records are entered into the dynamic simplified maptree table along with the time that they are detected (Line 9). This time is also broadcast throughout the network along with the record (Lines 10, 19).

**Module 3c** Dynamic adjacency graph propagation

1: Local variables: Dynamic simplified maptree table, $M_t = \langle rid_a : \mathbb{N}, rid_b : \mathbb{N}, cid : \mathbb{N}, t_1 : T, t_2 : T \rangle$, initialized with record $(-1, root, -1, 0, -1)$ where *root* is the id of the region component comprising the exterior of the region, Voronoi-junction table $J_t = \langle rid_a : \mathbb{N}, rid_b : \mathbb{N}, rid_c : \mathbb{N} \rangle$, initialized with zero records, Change table, $C_t = \langle w : \mathbb{N}, p_1 : \mathbb{N}, p_2 : \mathbb{N}, split : \{0,1\}, t : T \rangle$, initialized with zero records.

REGN

2: *When* adjacency timer elapsed
3: Reset adjacency, maptree timers
4: DELETE FROM $J_t$
5: **if** $\mathring{adj} = -1$ **then**
6: **let** *MinHop* := SELECT MIN $(h)$ FROM $B_t$
7: **let** *closest* := SELECT *bid* FROM $B_t$ WHERE $h = MinHop$
8: **if** $|closest| = 2$ **then**
9: INSERT INTO $M_t$ VALUES $(Min(closest), Max(closest), \varnothing, now, \varnothing)$
10: **broadcast** (bndy, *closest*, *now*)
11: **broadcast** (junq, $\mathring{s}(now)$, *closest*)
12: **if** $|closest| = 3$ **then**
13: INSERT INTO $J_t$ VALUES *closest*
14: **broadcast** (junf, *closest*)
15: *Receiving* (bndy, *pair*, $t'$)
16: **let** *record* := $(Min(closest), Max(closest), \varnothing, t', \varnothing)$
17: **if** *record* $\notin M_t$ **then**
18: INSERT INTO $M_t$ VALUES *record*
19: **broadcast** (bndy, *pair*, $t'$)
20: *Receiving* (junq, $s'$, *pair*)
21: **if** $\mathring{s}(now) = s'$ AND $adj \notin pair$ AND $adj \neq -1$ **then**
22: **let** *junction* := $pair \cup adj$
23: **if** *junction* $\notin J_t$ **then**
24: **broadcast** (junf, *junction*)
25: INSERT INTO $J_t$ VALUES *junction*
26: *Receiving* (junf, $J_t$)
27: **if** *junction* $\notin J_t$ **then**
28: INSERT INTO $J_t$ VALUES *junction*
29: **broadcast** (junf, *junction*)

### 5.1.5 *Module 4c: Simplified maptree generation for dynamic regions*

Once the maptree timer elapses, module 4 will assign connected components to each new entry in the simplified maptree table based on the previous time period's connected component members. The specifics of this can be broken into the following six stages:

1. Group region component ids into sets of connected components (Lines 2–10).

2. Find connected components that have merged, add record to change table ($C_t$), and assign new label to merged connected component (Lines 14–20).

3. Find connected components that have split, add to change table ($C_t$), and assign new labels to split connected components (Lines 21–28).

4. Assign labels to remaining connected components. Those that have a one or less difference in members from an old connected component will be assigned that connected component's label, and new connected components will be assigned a new label (Lines 1–8 pt. 2).

5. Close older simplified maptree records (Line 9 pt. 2).

6. Assign connected component ids to new records, remove duplicate records, and reopen any records where a duplicate has been removed (Lines 10-16 pt. 2).

Stage 1 operates identically to the original module from section 4.2. Stages 2 and 3 involve detecting split and merge events in the connected components between the current records and the previous records. To do this, a list of active connected component ids from the previous records (i.e., records that have not stored a value in $t_2$) is needed (Line 11), along with the time that the current records were opened (Line 12). A *members* function is also needed, which will return the set of region components that are Voronoi-adjacent to a connected component. Merge events occur when members of two connected components from the previous records are all present in a single connected component in the current records. For split events it is the reverse; members from a single connected component in the previous records now comprise two connected components in the current records.

These merge and split events are then added to the change table ($C_t$), and new ids are assigned to the newly split or merged connected components using the *clabel* function. Additionally, the region ids comprising these connected components, along with the new connected component ids, are added to the relations table ($R_t$). This is a temporary table that will be used in stage 6 to assign the correct connected component id to records in the simplified maptree table.

**Module 4c** Dynamic simplified maptree generation

1: Restrictions: component labeling function: $clabel(c) \rightarrow \mathbb{N}$ where $\mathbb{N}$ is a unique id for that set of region components, members function: $members(id) \rightarrow$ SELECT $rid_a, rid_b$ FROM $M_t$ WHERE $id = cid$ AND $t_2 = \varnothing$

REGN

2: *When* maptree timer elapsed
3: **let** $comp$ := *collection of sets derived from entries from* $J_t$
4: **for all** $i \in comp$ **do**
5: **for all** $j \in comp$ **do**
6: **if** $|i \cap j| \geq 2$ **then**
7: MERGE $(i, j)$
8: **let** *remaining* := *set of all unique region ids of open records from* $M_t$ *not present in comp*
9: **for all** $k \in remaining$ **do**
10: **let** $comp := comp \cup \{$SELECT $rid_a, rid_b$ FROM $M_t$ WHERE $t_2 = \varnothing$ AND $(rid_a = k$ OR $rid_b = k)\}$
11: **let** $cids :=$ SELECT $cid$ FROM $M_t$ WHERE $cid \neq \varnothing$ AND $t_2 = \varnothing$
12: **let** $t :=$ SELECT MAX$(t_1)$ FROM $M_t$
13: **let** $R_t = \langle components : set\ of\ rids, cid : \mathbb{N} \rangle$, initialized with zero records
14: **for all** $cids_i \in cids$ **do**
15: **for all** $cids_j \in cids$ **do**
16: **for all** $comp_k \in comp$ **do**
17: **if** $members(cids_i) \cup members(cids_i) = comp_k$ **then**
18: INSERT INTO $C_t$ VALUES $(clabel(comp_k), cids_i, cids_j, 0, t)$
19: INSERT INTO $R_t$ VALUES $(comp_k, clabel(comp_k))$
20: **set** $comp := comp \setminus \{comp_k\}$
21: **for all** $comp_i \in comp$ **do**
22: **for all** $comp_j \in comp$ **do**
23: **for all** $cids_k \in cids$ **do**
24: **if** $comp_i \cup comp_j = members(cids_k)$ **then**
25: INSERT INTO $C_t$ VALUES $(cids_k, clabel(comp_i), clabel(comp_j), 1, t)$
26: INSERT INTO $R_t$ VALUES $(comp_i, clabel(comp_i))$
27: INSERT INTO $R_t$ VALUES $(comp_j, clabel(comp_j))$
28: **set** $comp := comp \setminus \{comp_i, comp_j\}$

Stage 4 then looks through the remaining components that have not split or merged and attempts to match the current components with that of the previous connected components. If the set of region component ids differs by one or less member between the current records and the previous records, then the new component is assigned the label of the old component. This is done so that connected components won't be assigned new labels if a region component appears or disappears.

Stage 5 closes the records from the previous time step by filling their $t_2$ column with the time that the current records were opened. However, some of these records may continue, so in addition to assigning connected component ids to new records, stage 6 removes these duplicate records. Consider a simplified maptree table storing

a transition from region configuration 3 to region configuration 1 in Figure 27 at time period 10. Table 7 shows an example of this where stage 5 has been completed and stage 6 has assigned the correct connected component to the new record (entry 5). In this case entry 5 is a duplicate of entry 3, meaning that entry 5 can be removed and entry 3 can have its $t_2$ column set to $\varnothing$.

| $rid_a$ | $rid_b$ | $cid$ | $t_1$ | $t_2$ |
|---|---|---|---|---|
| -1 | 0 | -1 | 0 | $\varnothing$ |
| 0 | 1 | a | 0 | 10 |
| 0 | 2 | a | 0 | 10 |
| 1 | 2 | a | 0 | 10 |
| 0 | 2 | a | 10 | $\varnothing$ |

Table 7: Dynamic simplified maptree table illustrating duplicated records.

At this point, the algorithm is now capable of detecting any surrounds or engulfs relations at any previous time period using its dynamic simplified maptree table ($M_t$). By combing information from this table with the change table ($C_t$), this algorithm is also capable of detecting the specific type of transition that led to region components entering and exiting these relations.

**Module 4c** Dynamic simplified maptree generation (continued)

REGN

**for all** $comp_i \in comp$ **do**
  **let** *foundMatch* := FALSE
  **for all** $cids_j \in cids$ **do**
    **if** $|comp_i \triangle members(cids_j)| \leq 1$ **then**
      INSERT INTO $R_t$ VALUES ($comp_i$, $cids_j$)
      **set** *foundMatch* := TRUE
  **if** *foundMatch* = FALSE **then**
    INSERT INTO $R_t$ VALUES ($comp_i$, $clabel(comp_i)$)
UPDATE $M_t$ SET $t_2 = t$ WHERE $cid \neq \varnothing$ AND $t_2 = \varnothing$
**for all** $M_{ti} \in M_t | cid = \varnothing$ **do**
  **for all** $R_{tj} \in R_t$ **do**
    **if** $\{rid_a.M_{ti}, rid_b.M_{ti}\} \subseteq R_{tj}.components$ **then**
      UPDATE $M_{ti}$ SET $cid = R_{tj}.cid$
    **if** EXISTS SELECT $rid_a.M_t$, $rid_b.M_t$, $t_1.M_t$ FROM $M_t$ WHERE $rid_a.M_t = rid_a.M_{ti}$ AND $rid_b.M_t = rid_b.M_{ti}$ AND $cid.M_t = cid.M_{ti}$ AND $t_1.M_t < t_2.M_{ti}$ **then**
      DELETE FROM $M_t$ RECORD $M_{ti}$
      UPDATE $M_t$ SET $t_2 = \varnothing$ WHERE $rid_a.M_t = rid_a.M_{ti}$ AND $rid_b.M_t = rid_b.M_{ti}$ AND $cid.M_t = cid.M_{ti}$
UPDATE $M_t$ SET $t_2 = now$ WHERE $t_1 = t_2$

### 5.1.6 *Module 5c: Node movement for dynamic regions*

The final module is tasked with the detection of the topological events appear, disappear, merge, and split. The module then reruns modules 3 and 4 both periodically and when these events occur. Like the original module from section 4.2, module 5 refreshes the boundary table ($B_t$) (Lines 9–11) and updates both the region component and Voronoi region component ids by swapping them (Lines 5–7) whenever a node enters a new region (Line 3).

Additionally, nodes entering a region reset their regionChange timer and broadcast a request message to their neighbors (Lines 4, 12). Unlike previous timers, which are triggered at the same time for all nodes, this regionChange timer is specific to individual nodes. The regionChange timer is used to prevent nodes that have just changed regions from sending response messages (Lines 13–15). Preventing newly transitioned nodes from broadcasting information ensures that cases where multiple nodes enter a new region at the same time do not influence results.

Nodes that have sent a request message will then wait until they have received a response from all of their neighbors that have not recently transitioned. If all of the received sensed values match the node's own, then the region component the node was previously in has disappeared; this will be picked up by the next round of modules 3 and 4.

If all of the received sensed values do not match the node's own sensed value, then the region component the node has entered has just appeared. This new region component is then assigned a new component region id using the *rlabel* function, which produces an as yet unused id based on the current time and a region component id. Recall from the start of this chapter that at most one topological event can occur during a single time period; this ensures that the region will have a unique name. Additionally, if there are multiple nodes entering this new region component, they will all select the same id. The next round of modules 3 and 4 will then record this topological event in the dynamic simplified maptree ($M_t$) (Lines 16–19).

It is important to note that changes to the underlying region components must be gradual and continuous. Specifically, the boundaries of the underlying region components should not spontaneously expand or contract at a distance larger than the communication distance of the nodes. This expansion or contraction would be erroneously detected as a pair of simultaneous appearance and disappearance events.

**Module 5c** Dynamic node movement

1: Restrictions: region component labeling function: $rlabel(id, t) \rightarrow \mathbb{N}$ where $\mathbb{N}$ is a unique unused id for a region component based on an id and the current time.
2: Local variables: split component id $sid : V \rightarrow \mathbb{N} \cup \{-1\}$, initialized to $\mathring{id}$, split table $S_t = \langle rid : \mathbb{N}, sid : \mathbb{N} \rangle$.

REGN

3: *When* $\mathring{s}(now)$ *changes*
4: Reset regionChange timer
5: **set** $tmp := \mathring{rid}$
6: **set** $\mathring{rid} := \mathring{adj}$
7: **set** $\mathring{adj} := tmp$
8: **set** $\mathring{sid} := -1$
9: DELETE FROM $B_t$
10: **if** $\mathring{adj} \neq -1$ **then**
11: INSERT INTO $B_t$ VALUES ($\mathring{adj}$, 1)
12: **broadcast** (`rqst`, $\mathring{s}(now)$)
13: *Receiving* (`rqst`, $s'$)
14: **if** regionChange timer elapsed **then**
15: **broadcast** (`rspc`, $\mathring{s}(now)$)
16: *Receiving* (`rspc`, $s'$)
17: **if** ALL $s' \neq \mathring{s}(now)$ **then**
18: **set** $\mathring{rid} := rlabel(\mathring{adj}, now)$
19: **broadcast** (`hop`, $\mathring{s}(now)$, $\mathring{rid}$, 1)
20: *When* topologyChange timer elapsed
21: Reset topologyChange, split timers
22: **broadcast** (`merge1`, $\mathring{rid}$, $\mathring{s}(now)$)
23: DELETE FROM $S_t$
24: **set** $\mathring{sid} := \mathring{id}$
25: **broadcast** (`split1`, $\mathring{sid}$, $\mathring{s}(now)$)

While region component appearance and disappearance events can be detected by nodes polling the sensed values of their neighbors, the detection of merge and split events can only be detected by periodically checking for them. This is done using the topologyChange timer, which when elapsed sends out merge and split messages. To detect merge events, a `merge1` message is sent out with the node's current region component id (*rid*) along with its current sensed value (Line 22). If a neighboring region with the same sensed value has a different region component id, then two regions have merged (Lines 1–2 pt. 2). If this is the case, a new id is assigned using the *rlabel* function, the merge event is added to the node's change table ($C_t$), and a `merge2` message is broadcast (Lines 3–7 pt. 2). Nodes receiving this `merge2` message that have not stored this merge event will add it to their change table (Lines 8–11 pt. 2). Additionally, nodes that still have the region component id of the previous parts will update to the id of the new merged region (Lines 12–13 pt. 2).

Detecting the spiting of region components is done by first using leader election [8, 78, 81] to assign a split id (*sid*) to each current

region component (Lines 23–25 pt. 1, 14–17 pt. 2). This split id is then sent out along with the region component id, to be stored in the split table ($S_t$) of every node (Lines 18–23, 38 pt. 2). If there are records with the same region ids but different split ids, then a split has occurred in this region (Lines 24–26 pt. 2). This split is added to the change table and new region ids are assigned to the split region components (Lines 27–30 pt. 2). Additionally, if the current node is within one of these split region components, then their region component id is updated (Lines 31–37 pt. 2).

**Module 5c** Dynamic node movement (continued)

REGN

1: *Receiving* (`merge1`, $rid'$, $s'$)
2: **if** $\mathring{s}(now) = s'$ AND $\mathring{rid} \neq rid'$ **then**
3: **let** $p_1 :=$ MIN$(\mathring{rid}, rid')$
4: **let** $p_2 :=$ MAX$(\mathring{rid}, rid')$
5: **set** $\mathring{rid} := rlabel(\mathring{adj}, now)$
6: INSERT INTO $C_t$ VALUES $(\mathring{rid}, p_1, p_2, 0, now)$
7: **broadcast** (`merge2`, $\mathring{rid}$, $p_1$, $p_2$, $now$)
8: *Receiving* (`merge2`, $rid'$, $p_1'$, $p_2'$, $t'$)
9: **if** $(rid', p_1', p_2', 0, t') \notin C_t$ **then**
10: INSERT INTO $C_t$ VALUES $(rid', p_1', p_2', 0, t')$
11: **broadcast** (`merge2`, $rid'$, $p_1'$, $p_2'$, $t'$)
12: **if** $\mathring{rid} = p_1'$ OR $p_2'$ **then**
13: **set** $\mathring{rid} := rid'$
14: *Receiving* (`split1`, $sid'$, $s'$)
15: **if** $s' = \mathring{s}(now)$ AND $sid' < \mathring{sid}$ AND $\mathring{sid} \neq -1$ **then**
16: **set** $\mathring{sid} := sid'$
17: **broadcast** (`split1`, $\mathring{sid}$, $\mathring{s}(now)$)
18: *When* split timer elapsed
19: **if** $\mathring{sid} \neq -1$ **then**
20: **broadcast** (`split2`, $\mathring{rid}$, $\mathring{sid}$)
21: *Receiving* (`split2`, $rid'$, $sid'$)
22: **if** $(rid', sid') \notin S_t$ **then**
23: INSERT INTO $S_t$ RECORDS $(rid', sid')$
24: **for all** $S_{ti} \in S_t$ **do**
25: **for all** $S_{tj} \in S_t$ **do**
26: **if** $rid.S_{ti} = rid.S_{tj}$ AND $sid.S_{ti} = sid.S_{tj}$ **then**
27: **let** $p_1 := rlabel(sid.S_{ti}, now)$
28: **let** $p_2 := rlabel(sid.S_{tj}, now)$
29: **let** $w := rid.S_{ti}$
30: INSERT INTO $C_t$ VALUES $(w, p_1, p_2, 1, now)$
31: **if** $\mathring{rid} = rid.S_{ti}$ **then**
32: **if** $\mathring{sid} = -1$ **then**
33: **set** $\mathring{rid} := -1$
34: **if** $\mathring{sid} = sid.S_{ti}$ **then**
35: **set** $\mathring{rid} := p_1$
36: **if** $\mathring{sid} = sid.S_{tj}$ **then**
37: **set** $\mathring{rid} := p_2$
38: **broadcast** (`split2`, $rid'$, $sid'$)

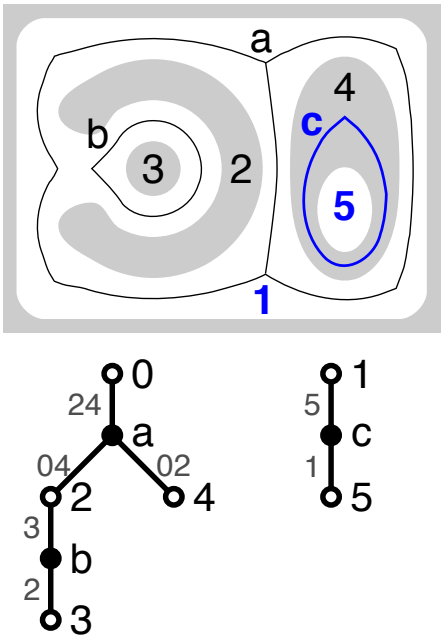
While the algorithms of the previous section are sufficient for determining the dynamic simplified maptree for simple regions, further modifications are required to accurately describe the dynamic simplified maptrees for complex areal objects. Of particular note is that region components are now capable of self merging and self-splitting. Consider the example dynamic region configuration shown in Figure 30. Like Figure 25 from section 4.3, this complex region example has two sets of Voronoi-adjacency boundaries, induced by the positive and the negative region components. Formally, $M_t^+$ and $M_t^-$ will refer to the simplified maptrees of the positive and negative region components respectively at a specific time period.

Between time periods 10 and 20 the region component 2 self-merges, causing region component component 2 to contain region component 3 instead of engulfing it. This also causes region component 1 to split into region components 6 and 7 and the appearance of connected component d. Storing these changes to the positive and negative simplified maptrees will be done with a slight modification to the dynamic simplified maptree table, namely the addition of a *neg* column that indicates whether the entry is part of the positive maptree or negative maptree. An example of this for Figure 30 is shown in table 8.

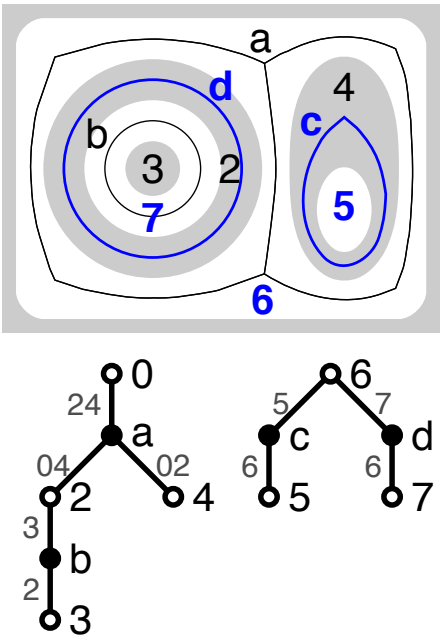
| $rid_a$ | $rid_b$ | $cid$ | $neg$ | $t_1$ | $t_2$ |
|---|---|---|---|---|---|
| -1 | 0 | -1 | 0 | 0 | $\varnothing$ |
| 0 | 2 | a | 0 | 0 | 40 |
| 0 | 4 | a | 0 | 0 | $\varnothing$ |
| 2 | 4 | a | 0 | 0 | 40 |
| 2 | 3 | b | 0 | 0 | 40 |
| 0 | 8 | a | 0 | 40 | $\varnothing$ |
| 4 | 8 | a | 0 | 40 | $\varnothing$ |
| 1 | 5 | c | 1 | 0 | 20 |
| 5 | 6 | c | 1 | 20 | 30 |
| 6 | 7 | d | 1 | 20 | $\varnothing$ |

Table 8: Simplified maptree table based on Figure 30.

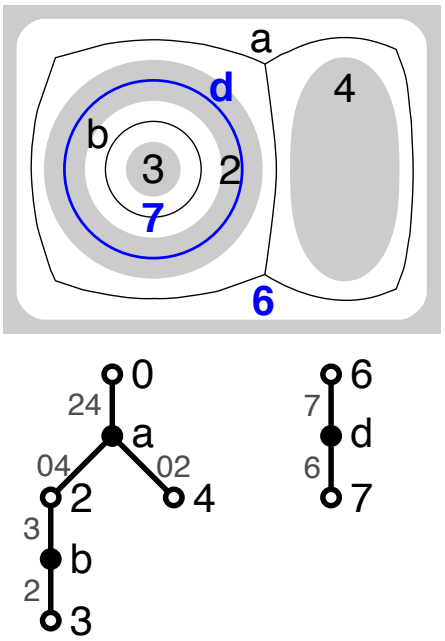
Like the dynamic simplified maptree table of the previous section, while this table is capable of showing the start and end times of region and connected components, it makes no distinction between appearance and split events, or between disappearance and merge events. Storing merge and split events will again be accomplished using the change table (table 9), which remains unchanged. Self-merge and self-split events are not explicitly stored as they coincide with splitting



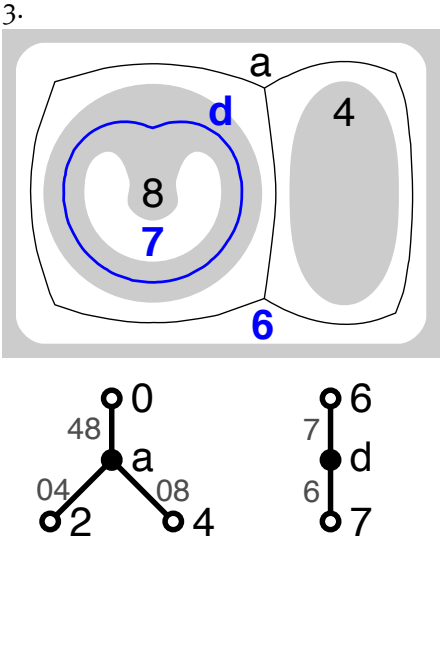

a. t=10, region component 2 engulfs region component 3.




b. t=20, region component 1 has split into regions 6 and 7, region component 2 now contains region 3.




c. t=30, region component 5 disappears.




d. t=40, regions components 2 and 3 have merged.

Figure 30: Dynamic region example with accompanying simplified maptree showing three time steps for a simple region. Positive region components are grey and negative region components are white. Black lines represent the Voronoi boundaries induced by positive region components and blue lines represent the Voronoi boundaries induced by negative region components.

and merging events of adjacent region components. For example, the self merging of the positive region component 2 occurs with the splitting of the negative region component 1 into region components 6 and 7.

| $w$ | $p_1$ | $p_2$ | *split* | $t$ |
|---|---|---|---|---|
| 1 | 6 | 7 | TRUE | 20 |
| 8 | 2 | 3 | FALSE | 30 |

Table 9: Change table ($C_t$) for logging split and merge events based on Figure 30.

In order to determine which region has self-merged or self-split from a split or merge event, information is required from the opposite simplified maptree. Specifically, the region component that has self-merged or self-split will be the one containing the connected component separating the connected components that have just split or merged. To use the splitting of the negative region component 1 as an example, it can be seen that region components 6 and 7 are separated by connected component d at this time period. As connected component d is contained within the positive region component 2 at this time period, it can be concluded that region component 2 has self-merged.

As mentioned in section 4.3, each connected component is contained within the Voronoi region component of its opposite simplified maptree; this relationship is defined by the *label* function. For this to work with dynamic simplified maptrees, this function must be extended to specify a time period, i.e., $B_t^+ \rightarrow W_t^-$. This function can be stored in a label table ($L_t$), and an example output of this for the object in Figure 30 is presented in Table 10.

| *cid* | *rid* | $t_1$ | $t_2$ |
|---|---|---|---|
| a | 1 | 0 | 20 |
| a | 6 | 20 | $\varnothing$ |
| b | 1 | 0 | 20 |
| b | 7 | 20 | 40 |
| c | 4 | 0 | 30 |
| d | 2 | 20 | 40 |
| d | 8 | 40 | $\varnothing$ |

Table 10: Label table ($L_t$) for logging mappings between conected components and the region components that contain them based on Figure 30.

### 5.2.1 *Qualitative relations*

By extending the dynamic data structure to account for complex regions, the qualitative relation detection can be further extend to additionally cover the ways in which region components both enter and leave the contains relation. Looking at the conceptual neighborhood graph for complex regions shown in Figure 31, there are three ways to transition to or from the contains relation. This is in contrast to the four possible transitions for the engulfs and surrounds relations.

Looking at the three categories of transitions, though the first and second categories (where a region component appears or disappears and where a region component splits or merges) are represented, the third category is not. Recall that the third category is for transitions where there is no topological change but a change in the internal configuration of the regions, as demonstrated by the capture and ensnare transitions. This category of transitions is not possible for the contains relation as it would involve the contained region component pushing through the containing region.

Again for simplicity's sake, only transitions entering the contains relation will be discussed as detecting the reverse of these transitions can be done by reversing the order of detected events. Additionally, transitions to and from the engulfs and surrounds relations will not be discussed as there is no change to their function from the previous section.

Another case where region component appearance creates a qualitative relation, is the transition between region configuration 7 and the contains relation (appear). As was the case for the engulf and surround transitions, a new region has appeared. This appear transition is most like the engulf transition as it also features the appearance of a new connected component (b). This transition can be found by checking for the presence of a new contains relation and connected component after the appearance of a new region component.

For split events, there are two additional cases where a qualitative relation is created; the transition between region configuration 8 and the contains relation (split), and the transition between the engulfs and contains (confine) relation. In the case of split events, the region component has split to form the containing and contained region components. This is in contrast to the confine transition where the negative region component (9) has split, causing the engulfing region component to self-merge. Further distinguishing these two transitions is the appearance of a connected component (b) for the split transition and the appearance of a negative connected component (e) for the confine transition. The split transition can be found by checking for the presence of a new contains relation after a split event in a positive region component and the appearance of a new connected component. This is in contrast to the confine transition, which must

instead be accompanied by the splitting of a negative region component and the appearance of a negative connected component.

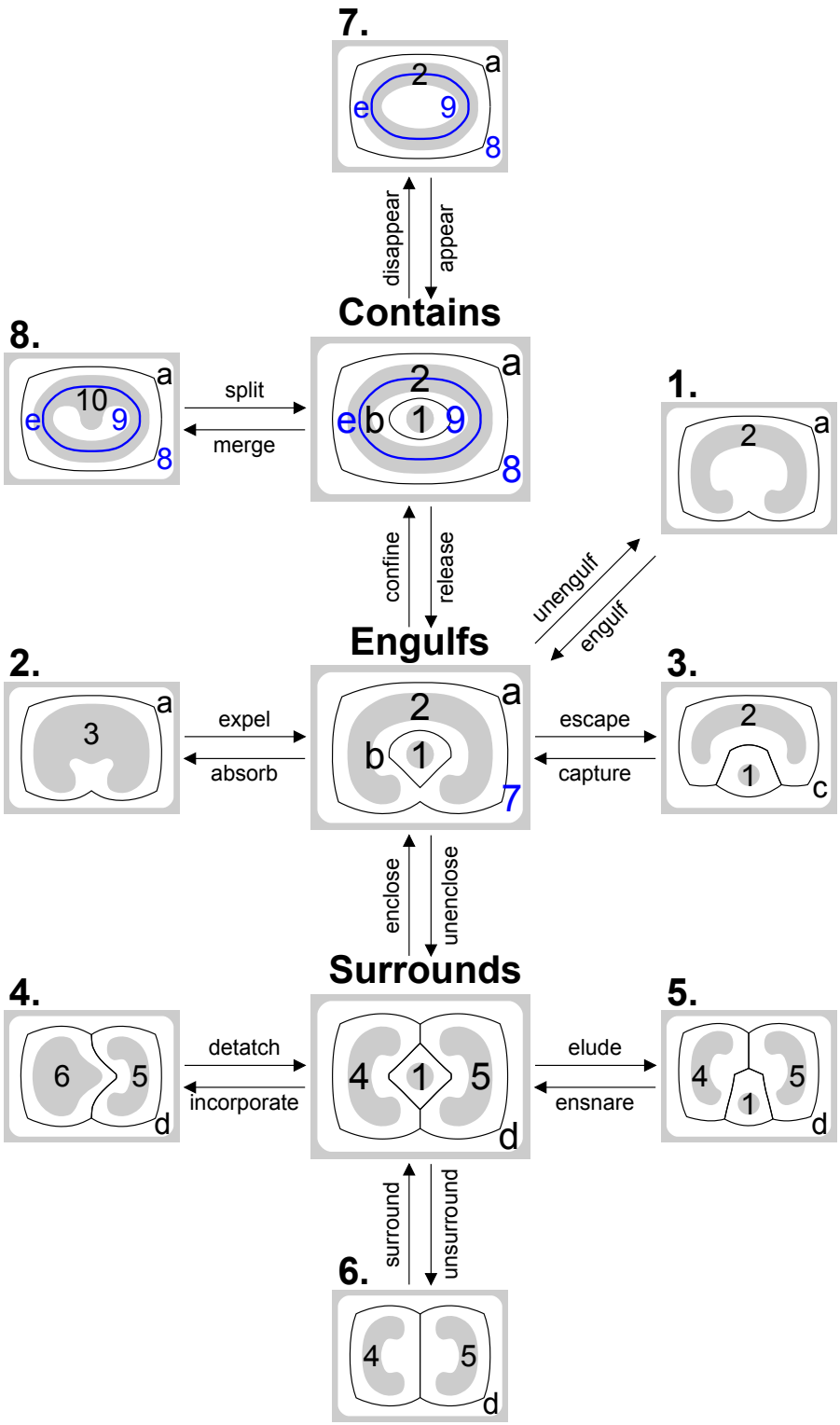



Figure 31: Conceptual neighborhood graph for complex regions showing the methods by which region components can configure into or out of an contains, engulfs or surrounds relation.

### 5.2.2 *Algorithm design*

To extend the implementation of the dynamic simplified maptree for simple regions to complex regions, some key changes from the previous algorithm must be made. Specifically, a label table, $L_t$, must be added to store what region components the connected components reside within at any given time. This is for the extended *label* function to be able to distinguish between engulfs and contains relations as well as to detect self-split and self-merge events. Additionally, a new column must be added to the simplified maptree table ($M_t$) in order to distinguish entries that are part of the positive and negative dynamic simplified maptrees. To implement these new features, it is only necessary to update modules 3 and 4. This algorithm will be referred to as the complexDynamic algorithm, and the relations between the modified modules are identical to those shown for the simpleDynamic algorithm in Figure 29. To explain the specifics of implementing this model, the following sections, which include a description of the modified modules and their pseudocode, have been provided.

### 5.2.3 *Module 3d: Adjacency relation identification for complex dynamic regions*

Module 3 runs almost identically to the module from section 5.1, however where this module differs is that the dynamic simplified maptree table,$M_t$, has an additional *neg* column to indicate whether a relation is a part of the positive or negative dynamic simplified maptree table. This *neg* column is populated with the sensed value of the node that has detected the relation (Line 9).

Additionally, the adjacency table, $A_t$, has been added to store the ids of region component pairs that are adjacent. This table is cleared each time module 3 is run (Lines 1, 4). Nodes on the boundary between two Voronoi regions will add the id of these regions along with their region component id (*rid*) to their adjacency table before broadcasting this information throughout the network using the `bndy` message (Lines 8, 10–12). Using Figure 30.a as an example, a node on the boundary between the Voronoi region components 2 and 4 would add the records $(1,2)$ and $(1,4)$ to its adjacency table. In addition to storing new simplified maptree records, nodes receiving a `bndy` message will add the received region component ids to their adjacency tables before rebroadcasting the message (Lines 17, 21-23). This table will be used by module 4 to determine which region component a connected component resides within.

**Module 3d** Dynamic adjacency graph propagation for complex regions

1: Local variables: Dynamic simplified maptree table, $M_t = \langle rid_a : \mathbb{N}, rid_b : \mathbb{N}, cid : \mathbb{N}, neg : \{0,1\}, t_1 : T, t_2 : T\rangle$, initialized with record $(-1, root, -1, 0, 0, -1)$ where *root* is the id of the region component comprising the exterior of the region, Voronoi-junction table $J_t = \langle rid_a : \mathbb{N}, rid_b : \mathbb{N}, rid_c : \mathbb{N}\rangle$, initialized with zero records, Change table, $C_t = \langle w : \mathbb{N}, p_1 : \mathbb{N}, p_2 : \mathbb{N}, split : \{0,1\}, t : T\rangle$, initialized with zero records, Adjacency table $A_t = \langle rid_a : \mathbb{N}, rid_b : \mathbb{N}\rangle$, initialized with zero records.

REGN

2: *When* adjacency timer elapsed
3: Reset adjacency, maptree timers
4: DELETE FROM $J_t$, $A_t$
5: **if** $\mathring{adj} = -1$ **then**
6: **let** *MinHop* := SELECT MIN $(h)$ FROM $B_t$
7: **let** *closest* := SELECT *bid* FROM $B_t$ WHERE $h = MinHop$
8: **if** $|closest| = 2$ **then**
9: INSERT INTO $M_t$ VALUES $(Min(closest), Max(closest), \varnothing, \mathring{s}(now), now, \varnothing)$
10: INSERT INTO $A_t$ VALUES $(Min(\mathring{rid}, Min(closest)), Max(\mathring{rid}, Min(closest)))$
11: INSERT INTO $A_t$ VALUES $(Min(\mathring{rid}, Max(closest)), Max(\mathring{rid}, Max(closest)))$
12: **broadcast** (bndy, *closest*, $\mathring{rid}$, $\mathring{s}(now)$, *now*)
13: **broadcast** (junq, $\mathring{s}(now)$, *closest*)
14: **if** $|closest| = 3$ **then**
15: INSERT INTO $J_t$ VALUES *closest*
16: **broadcast** (junf, *closest*)
17: *Receiving* (bndy, *pair*, *rid*′, *s*′, *t*′)
18: **let** *record* := $(Min(closest), Max(closest), \varnothing, s', t', \varnothing)$
19: **if** *record* $\notin M_t$ **then**
20: INSERT INTO $M_t$ VALUES *record*
21: INSERT INTO $A_t$ VALUES $(Min(rid', Min(closest)), Max(rid', Min(closest)))$
22: INSERT INTO $A_t$ VALUES $(Min(rid', Max(closest)), Max(rid', Max(closest)))$
23: **broadcast** (bndy, *pair*, *rid*′, *s*′, *t*′)
24: *Receiving* (junq, *s*′, *pair*)
25: **if** $\mathring{s}(now) = s'$ AND $adj \notin pair$ AND $adj \neq -1$ **then**
26: **let** *junction* := $pair \cup adj$
27: **if** *junction* $\notin J_t$ **then**
28: **broadcast** (junf, *junction*)
29: INSERT INTO $J_t$ VALUES *junction*
30: *Receiving* (junf, $J_t$)
31: **if** *junction* $\notin J_t$ **then**
32: INSERT INTO $J_t$ VALUES *junction*
33: **broadcast** (junf, *junction*)

### 5.2.4 *Module 4d: Simplified maptree generation for complex dynamic regions*

As before, module 4 will group the present detected adjacency components into connected components based on the previous time period's connected component members. The specifics of this were covered by the following six stages used by the previous algorithm:

1. Group region component ids into sets of connected components (Lines 2–10).
2. Find connected components that have merged, add record to change table ($C_t$), and assign a new label to the merged connected component (Lines 14–20).
3. Find connected components that have split, add to change table, and assign new labels to split connected components (Lines 21–28).
4. Assign labels to remaining connected components. Those with one or fewer difference in members from an old connected component will be assigned that connected component's label. New connected components will be assigned a new label (Lines 1–8 pt. 2).
5. Close older simplified maptree records (Line 9 pt. 2).
6. Assign connected component ids to new records, remove duplicate records, and reopen any records where a duplicate has been removed (Lines 10-16 pt. 2).

Where this module differs is with the addition of the following two stages, which keep the label table ($L_t$) updated:

7. Close older label records (Line 17 pt. 2).
8. Match connected component ids to their region components, either making new records or reopening unchanged records (Lines 18-23 pt. 2).

Stage 6 closes the label records from the previous time step by filling their $t_2$ column with the time that the current records were opened. Some of these records may continue, so in addition to matching connected component ids with region component ids, stage 7 reopens these unchanged records. Consider the label table storing a transition from Figure 30.a to Figure 30.b; table 11 shows an example of this where stage 6 has closed the first three records but stage 7 has has had to reopen the third record.

To match the connected component ids with region component ids, module 4 requires the adjacency function, which returns the set of adjacent regions for any given region. Running this function on region

| *cid* | *rid* | $t_1$ | $t_2$ |
|---|---|---|---|
| a | 1 | 10 | 20 |
| b | 1 | 10 | 20 |
| c | 4 | 10 | $\varnothing$ |
| a | 6 | 20 | $\varnothing$ |
| b | 7 | 20 | $\varnothing$ |
| d | 2 | 20 | $\varnothing$ |

Table 11: Label table of Figure 30 when $t = 20$.

component 6 of Figure 30.b would produce $adjacent(6) = \{0, 2, 4\}$, meaning that region component 6 is adjacent to region components 0, 2, and 7. Given that each connected component is stored along with its set of region component ids, it is then possible to perform a match. For example, in Figure 30.b it can be determined that connected component $a$ resides within region component 6, given that $a$ consists of region components 0, 2, and 4, which is an improper subset of $adjacent(6)$ (Line 19).

At this point, the algorithm is now capable of detecting any contains, surrounds, or engulfs relations at any previous time period using its dynamic simplified maptree table ($M_t$). By combing information from this table with the change table ($C_t$), this algorithm is also capable of detecting the specific type of transition that led to region components entering and exiting these relations.

**Module 4d** Dynamic simplified maptree generation for complex regions

1: Restrictions: component labeling function: $clabel(c) \rightarrow \mathbb{N}$ where $\mathbb{N}$ is a unique id for that set of region components, members function: $members(id) \rightarrow$ SELECT $rid_a, rid_b$ FROM $M_t$ WHERE $id = cid$ AND $t_2 = \varnothing$, adjacency function: $adjacent(rid) \rightarrow \{rid' \in R | (rid, rid') \in A_t\}$

REGN

2: *When* maptree timer elapsed
3: **let** $comp$ := *collection of sets derived from entries from* $J_t$
4: **for all** $i \in comp$ **do**
5: **for all** $j \in comp$ **do**
6: **if** $|i \cap j| \geq 2$ **then**
7: MERGE $(i, j)$
8: **let** *remaining* := *set of all unique region ids of open records from* $M_t$ *not present in comp*
9: **for all** $k \in remaining$ **do**
10: **let** $comp := comp \cup \{$SELECT $rid_a, rid_b$ FROM $M_t$ WHERE $t_2 = \varnothing$ AND $(rid_a = k$ OR $rid_b = k)\}$
11: **let** $cids :=$ SELECT $cid$ FROM $M_t$ WHERE $cid \neq \varnothing$ AND $t_2 = \varnothing$
12: **let** $t :=$ SELECT MAX$(t_1)$ FROM $M_t$
13: **let** $R_t = \langle components$ : *set of rids*, $cid : \mathbb{N} \rangle$, initialized with zero records
14: **for all** $cids_i \in cids$ **do**
15: **for all** $cids_j \in cids$ **do**
16: **for all** $comp_k \in comp$ **do**
17: **if** $members(cids_i) \cup members(cids_i) = comp_k$ **then**
18: INSERT INTO $C_t$ VALUES $(clabel(comp_k), cids_i, cids_j, 0, t)$
19: INSERT INTO $R_t$ VALUES $(comp_k, clabel(comp_k))$
20: **set** $comp := comp \setminus \{comp_k\}$
21: **for all** $comp_i \in comp$ **do**
22: **for all** $comp_j \in comp$ **do**
23: **for all** $cids_k \in cids$ **do**
24: **if** $comp_i \cup comp_j = members(cids_k)$ **then**
25: INSERT INTO $C_t$ VALUES $(cids_k, clabel(comp_i), clabel(comp_j), 1, t)$
26: INSERT INTO $R_t$ VALUES $(comp_i, clabel(comp_i))$
27: INSERT INTO $R_t$ VALUES $(comp_j, clabel(comp_j))$
28: **set** $comp := comp \setminus \{comp_i, comp_j\}$

**Module 4d** Dynamic simplified maptree generation for complex regions (continued)

REGN

1: **for all** $comp_i \in comp$ **do**
2: **let** *foundMatch* := FALSE
3: **for all** $cids_j \in cids$ **do**
4: **if** $|comp_i \triangle members(cids_j)| \leq 1$ **then**
5: INSERT INTO $R_t$ VALUES ($comp_i$, $cids_j$)
6: **set** *foundMatch* := TRUE
7: **if** *foundMatch* = FALSE **then**
8: INSERT INTO $R_t$ VALUES ($comp_i$, $clabel(comp_i)$)
9: UPDATE $M_t$ SET $t_2 = t$ WHERE $cid \neq \varnothing$ AND $t_2 = \varnothing$
10: **for all** $M_{ti} \in M_t | cid = \varnothing$ **do**
11: **for all** $R_{tj} \in R_t$ **do**
12: **if** $\{rid_a.M_{ti}, rid_b.M_{ti}\} \subseteq R_{tj}.components$ **then**
13: UPDATE $M_{ti}$ SET $cid = R_{tj}.cid$
14: **if** EXISTS SELECT $rid_a.M_t$, $rid_b.M_t$, $t_1.M_t$ FROM $M_t$ WHERE $rid_a.M_t = rid_a.M_{ti}$ AND $rid_b.M_t = rid_b.M_{ti}$ AND $cid.M_t = cid.M_{ti}$ AND $t_1.M_t < t_2.M_{ti}$ **then**
15: DELETE FROM $M_t$ RECORD $M_{ti}$
16: UPDATE $M_t$ SET $t_2 = \varnothing$ WHERE $rid_a.M_t = rid_a.M_{ti}$ AND $rid_b.M_t = rid_b.M_{ti}$ AND $cid.M_t = cid.M_{ti}$
17: UPDATE $L_t$ SET $t_2 = t$ WHERE $t_2 = \varnothing$
18: **for all** $R_{ti} \in R_t$ **do**
19: **let** *conatinedIn* := $rid | \{components.R_{ti} \subseteq adjacent(rid)\}$
20: **if** EXISTS SELECT $*$ FROM $L_t$ WHERE $cid = cid.R_{ti}$ AND $rid = containedIn$ AND $t_2 = t$ **then**
21: UPDATE $L_t$ SET SET $t_2 = \varnothing$ WHERE $cid = cid.R_{ti}$ AND $rid = containedIn$ AND $t_2 = t$
22: **else**
23: INSERT INTO $L_t$ VALUES ($cid.R_{ti}$, *containedIn*, $t$, $\varnothing$)
24: UPDATE $M_t$ SET $t_2 = now$ WHERE $t_1 = t_2$

## 5.3 SUMMARY

This chapter has presented two decentralized algorithms that are capable of determining the internal structure of dynamic regions and discovering any qualitative relations that may be present as well as the manner in which these relations were entered into and exited from. Like the algorithms of the previous chapter, this is all done without reference to location and while the nodes within the network are mobile. The key difference between the algorithms of this chapter and those of the previous is that these algorithms are capable of detecting topological events occurring in the underlying region and recording these changes. While both algorithms of this chapter are capable of extracting high level knowledge from low level sensor data provided by individual nodes in the network, there are some key differences between them:

- The first algorithm is only capable of running on simple regions and introduces the dynamic simplified maptree data structure for the storage of the adjacency relations of the Voronoi regions induced by the region components. This algorithm is capable of detecting region component appearance, disappearance, splitting, and merging. Additionally, this algorithm is capable of detecting the four distinct ways a region component can enter and exit a surrounds or engulfs relation.

- The second algorithm extends the first so that it may run on complex region configurations. It does so by splitting the dynamic simplified maptree into two maptrees; the positive simplified maptree for relations induced by the positive region components and the negative simplified maptree for the relations induced by the negative region components. This algorithm is additionally capable of detecting region component self-merge and self-split, and can distinguish between engulfs and contains relations. Furthermore, this algorithm is capable of detecting the three distinct ways a region component can enter and exit a contains relation.

Chapter 6 will evaluate these algorithms in terms of communication complexity and veracity.

# 6

# EVALUATION

This chapter presents experimental evaluations of the three algorithms devised for static regions presented in Chapter 4 and the two algorithms devised for dynamic regions presented in Chapter 5. As discussed in Chapter 3, these algorithms have been implemented in the NetLogo simulation environment, with their performance evaluated in terms of veracity and then scalability. For ease of reference, the names of these five algorithms will be reiterated here:

1. **basicStatic:** Basic data structure algorithm from section 4.1, capable of running on simple or complex static regions.
2. **simpleStatic:** Simplified maptree algorithm from section 4.2, capable of running on simple static regions.
3. **complexStatic:** Simplified maptree algorithm from section 4.3, capable of running on simple or complex static regions.
4. **simpleDynamic:** Dynamic simplified maptree algorithm from section 5.1, capable of running on simple dynamic regions.
5. **complexDynamic:** Dynamic simplified maptree algorithm from section 5.2, capable of running on simple or complex dynamic regions.

To illustrate the specifics of the interactions between the modules comprising the five algorithms, Figure 32 has been provided.

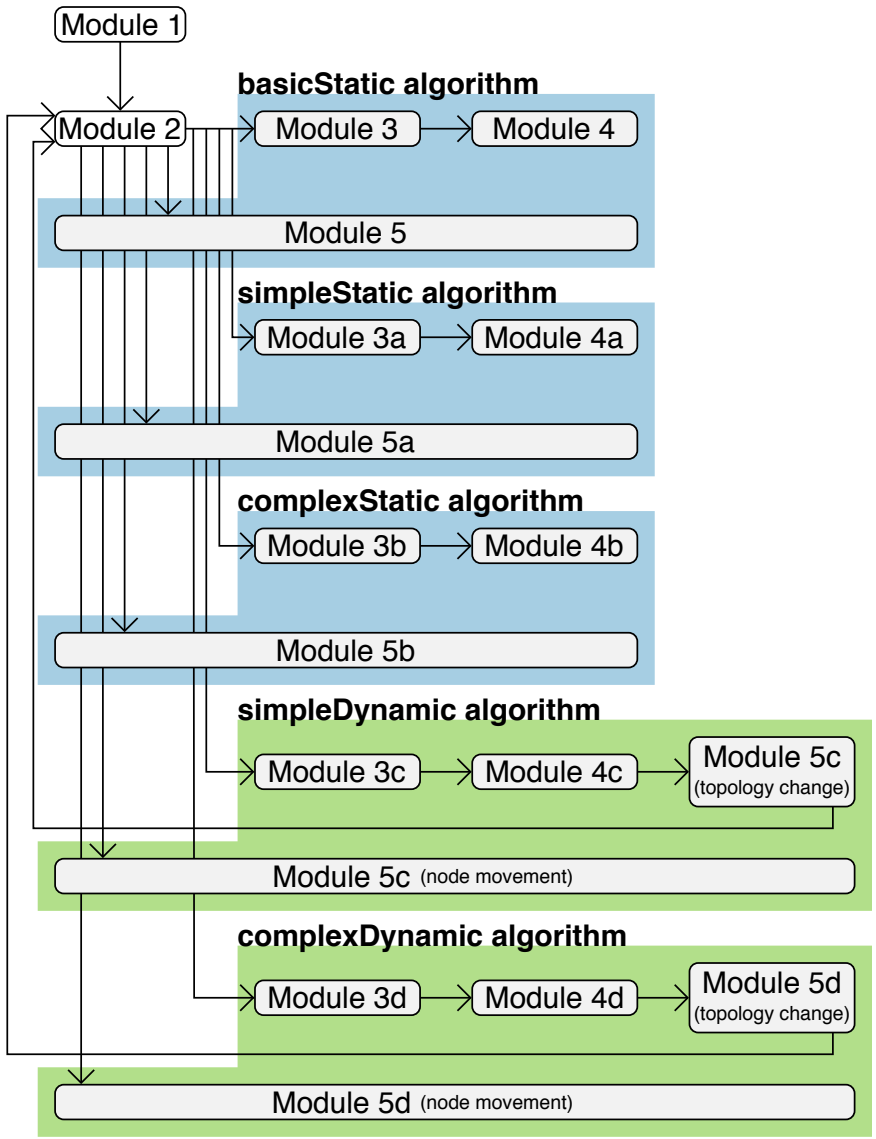



Figure 32: Flow diagram representing the interactions between the modules that comprise the five algorithms. Modules shaded blue comprise algorithms designed for static regions and modules shaded green comprise algorithms designed for dynamic regions.

During these evaluations, nodes within the networks move according to a Correlated Random Walk (CRW) [82] over a variety of region configurations within the simulation environment. A node moving according to a CRW exhibits a correlation in its heading between subsequent movement steps. As this correlation increases, the node tends towards moving in a straight line. This is in contrast to random walks, where there is no correlation between subsequent headings. As all di-

rections are equally likely for random walks, nodes would instead move according to Brownian motion.

A CRW movement pattern has been chosen for these nodes as it provides a simple approximation of many natural movement patterns, such as those observed in herd animals [83, 84]. This movement pattern was implemented by nodes selecting a new heading from a normal distribution with a mean centered at the node's current heading and a standard deviation of 45°. Nodes would then move a distance of one patch width in that direction. Additionally, nodes were assigned a random location and heading at the start of each simulation.

## 6.1 VERACITY

Common to all five algorithms is that they utilize the same code for modules 1 and 2. Assuming that the conditions are met for these modules to perform correctly, all five algorithms will produce correct results. Figure 33 shows an example implementation of modules 1 and 2 on a complex areal object originally displayed in Figure 25.

To examine veracity, a simulation environment consisting of four square alternating positive and negative region components was used. Each region was 20 patches (a square of arbitrary size used in NetLogo) in width giving a total simulation size of 80×20 patches. This simulation environment wrapped both horizontally and vertically (i.e., projected onto a torus) to eliminate any edge effects (i.e., issues that occur when nodes reach the edge of the simulation space).

### 6.1.1 *Veracity of module 1*

When discussing the veracity of an algorithm, it is important to first define the conditions the algorithm will function under. Module 1 assumes that there will be at least one node in each region component, with all nodes in that region component making up a connected subgraph. It is further assumed that the communication distance will be smaller than the smallest distance between any two positive or any two negative region components. If the distance is larger, then it is possible that two separate regions will be falsely counted as one; this is a type of error known as a granularity effect.

To test these assumptions, module 1 was run for 30 ticks (simulation time steps) under a variety of network sizes and communication distances, recording the percentage of nodes with the correct region component id. Six network sizes were chosen as well as communication distances between 1 and 30 patch widths. This module was run for 30 ticks as it is larger than the diagonal distance of the region components (28.28). Given that the smallest communication distance

Figure 33: Expected and observed regions as well as Voronoi boundaries for the complex areal object originally displayed in Figure 25. Black and white lines represent the Voronoi boundaries induced by positive and negative region components respectively. Displays an example implementation of modules 1 and 2 on a network with 10,000 nodes.

is 1, it would take at most 28 ticks to send a message from one corner of a region component to its diagonal opposite.

The average of 100 runs for each of these configurations was used, leading to a total of 18,000 experimental runs. Recall that each region is a square of $20 \times 20$ patches so any communication distance over 20 patches will lead to two region components sharing the same id. When two region components share the same id, all nodes within those regions were logged as having an incorrect region component id.

From Figure 34, it can be seen that there is poor performance at low communication distances for all network sizes, meaning that the network is too sparsely populated for all nodes within a region component to be connected. The performance of larger networks improves faster with increasing communication distance but drops rapidly after a distance of 20 patches for all network sizes. This drop is due to

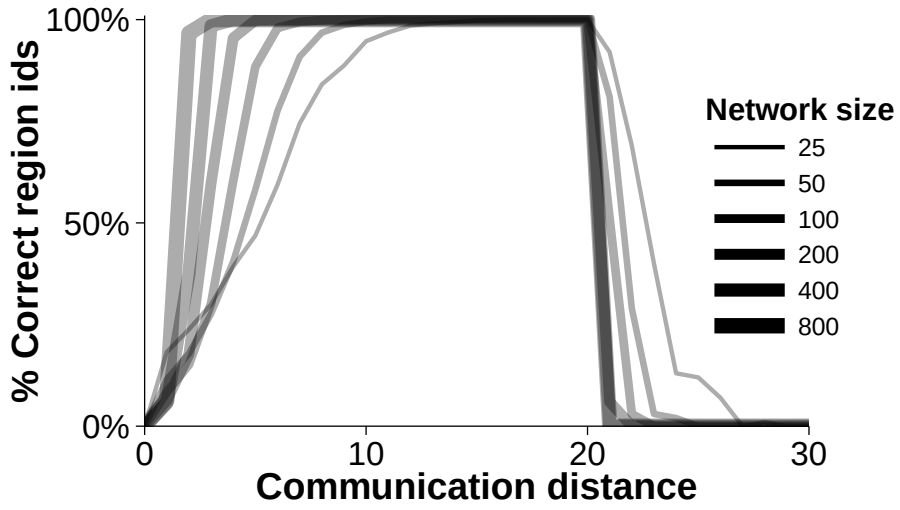



Figure 34: Veracity of module 1 for varying network sizes and communication distances, adapted from [12].

the effect of network granularity, which causes multiple region components to share the same id.

From this graph, it can be concluded that both communication distance and network size influence module 1's performance, with communication distance having the greater effect. This experiment illustrates the type of trade-offs that occur when using decentralized approaches. In this case, it is between network size and communication distance. For example, a network size of 25 and a communication distance of 14 produces results that are as accurate as those produces by a network size of 800 and a communication distance of 3. Specifically, as transmission costs increase rapidly with communication distance, it may be more cost effective to increase the network size and decrease the communication distance.

### 6.1.2 *Veracity of module 2*

Common to module 5 for every algorithm is that when a node enters a new region component (i.e., its sensed value has changed), it swaps its region component id and Voronoi region component id (*rid* and *adj* respectively). For any of these algorithms to function correctly, a node must know the correct id of the Voronoi region component it is in, as well as the correct region component id prior to changing regions. For this to occur, nodes traveling between positive or between negative regions must have their Voronoi region id updated after they have crossed the Voronoi boundary but before they have changed regions.

To ensure that the Voronoi boundary is in the correct location when using a mobile geosensor network, module 2 must be rerun periodically. How often this module is rerun is determined by the length of

the broadcast timer. Assuming that the communication speed of module 2 is much faster than the movement speed of the nodes, how often module 2 must be run can be determined. Suppose $d$ is the shortest distance between any two positive or negative region components, $s$ is the maximum speed of the nodes, and $b$ is the broadcast interval of module 2. The following formula can predict the largest refresh interval needed to ensure that the algorithm runs correctly: $b \leq \frac{d}{2s}$.

For this experiment, the region configuration is unchanged, meaning that the minimum distance between any two positive or negative region components is 20 patches. Setting the node speed to 1 patch per tick, it can be assumed that any refresh interval up to once every 10 ticks will be capable of producing correct results. This is assuming that the node density is high enough that nodes will cross a Voronoi boundary between either positive or negative region components. This experiment has been run with four network sizes and broadcast intervals between 1 and 50. Again, the average of 100 runs for each of these configurations was used, leading to a total of 20,000 experimental runs. Each run allowed for 10 rounds of module 2 (i.e.: that the `hop` message was sent 10 times). Communication distance was set at 5 patch widths and nodes moved according to a correlated random walk.

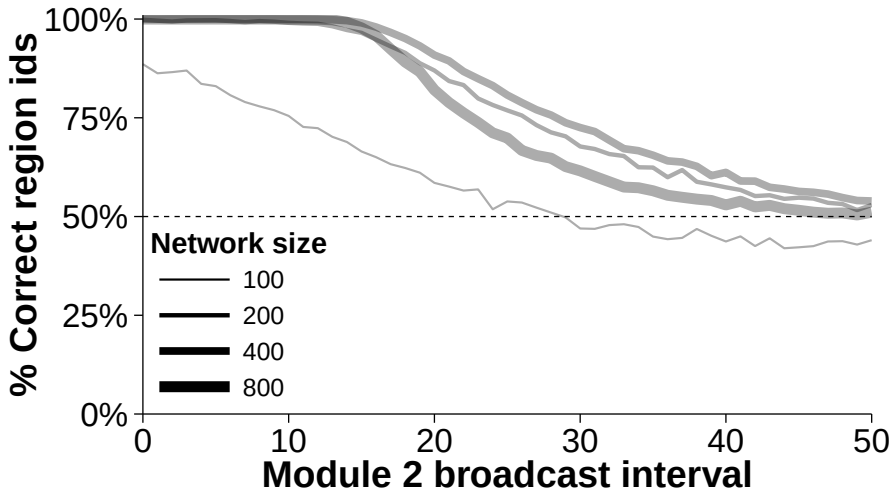



Figure 35: Veracity of module 2 for varying network sizes and communication distances.

From Figure 35, it can be seen that given a sufficient node density (in this case, a network size of 200 or greater), module 2 will produce correct results up to a refresh interval of approximately 12, after which the accuracy drops to approximately 50% at varying rates based on network size. This was expected as there are two positive and two negative region components, meaning that there is still a 50% chance of a node having the correct region id regardless of module 2's accuracy.

Looking at the results for a network size of 100, it can be seen that no refresh interval will produce completely accurate results. This, like the low communication distances in Figure 34, is due to the network being too sparsely populated for all nodes within a region component to be connected, leading to the generation of superfluous region component ids. Specifically, network sizes of 100 detected an average of 8.16 region components. This was also an issue to a lesser extent for networks of size 200, which detected an average of 4.2 region components. However, this network size was still able to produce very accurate results ($> 99\%$ for refresh intervals $\leq 10$), indicating that these superfluous regions contained very few nodes. Networks with 400 and 800 nodes were of sufficient density to always detect precisely four region components.

It is interesting to note that the refresh interval producing correct results was larger than the calculated interval of 10. This difference is expected as the formula assumes that nodes will follow the shortest path between positive or between negative region components; however this is not the case in the experiments as the nodes move according to a correlated random walk. From this graph, it can be concluded that while network size and the refresh interval of module 2 both influence module 5's performance, selecting a refresh interval close to the calculated refresh interval, and a node density that allows sufficient communication of module 2's message, will always produce accurate results.

## 6.2 SCALABILITY OF STATIC REGIONS

As discussed in section 3.3, an algorithm's scalability is determined by how the amount of node communication is affected by an increase in network size. Scalability can be divided into communication complexity when considering the amount of communication necessary for the entire network, and load balance when considering the amount of communication for individual nodes.

The algorithms tested in this section ran on either the simple region configuration of Figure 22 or the complex region configuration of Figure 25. In either case, this produced a simulation size of $290 \times 145$ patches. Communication distance was initially set to a radius of ten patch widths, giving nodes an average communication neighborhood of approximately 30 nodes for a network size of 4,000.

Node movement occurred every 50 ticks, with a movement distance of 2.5 patches. The broadcast timer was set to 25, meaning that module 2 was rerun twice as often as the nodes moved. The region timer was set to 100, giving 100 ticks for module 1 to initialize the network. The adjacency timer was set to 170, meaning that module 3 ran after 170 ticks. The maptree timer was set for 50 ticks later (at 220), giving 50 ticks for module 3 to complete before module 4 began. Ad-

ditionally, the containment message for the basicStatic algorithm was injected at 220 ticks. To aid readability of the graphs, the number of messages sent by module 2 was divided by the number of broadcast rounds. This means that the number of messages plotted for module 2 represents the average number of messages sent for a single broadcast round.

After 350 ticks, the total number of messages sent by each module for the entire network was recorded (i.e., communication complexity) and the maximum number of messages sent by an individual node for each module (i.e., load balance) was recorded. This was done 100 times with randomized node positions for each network size (4,000, 6,000, 8,000, and 10,000 nodes), leading to a total of 400 experimental runs for each tested algorithm and region configuration.

### 6.2.1 *Simple region configuration*

Looking at the simple region configuration of Figure 22, approximately 47% of the area is covered by the nine positive region components and 53% is covered by the single negative region component. This region configuration consists of 14 unique Voronoi region boundaries and seven unique Voronoi junctions.

#### *basicStatic algorithm*

Figure 36 shows the communication complexity of the basicStatic algorithm on a simple region configuration. All regression curves for each of the modules bar module 3 achieved a good fit, as evidenced by $R^2$ values of greater than 0.98. Module 3 achieved a comparatively lower value of 0.91, indicating greater variation between experimental runs.

From this graph, it can be inferred that modules 3 and 4 scaled linearly in terms of communication complexity whereas modules 1, 2, and 5 were of polynomial order. This was expected for modules 3 and 4 as they are based on surprise flooding algorithms, which scale linearly based on the number of unique messages that are sent. For module 4, a single message must be sent to each node in the network meaning that the number of messages sent will always equal the number of nodes in the network, regardless of the configuration of the underlying region.

For module 3, a message must be sent for each unique Voronoi region boundary (of which there are 14 in the simple region configuration) to each node in the region component that contains these boundaries. As Voronoi region boundaries are only contained within the single negative region component for this region configuration, and approximately half of the network's nodes will be within this region component, it is logical that approximately seven messages were sent for each node in the network.

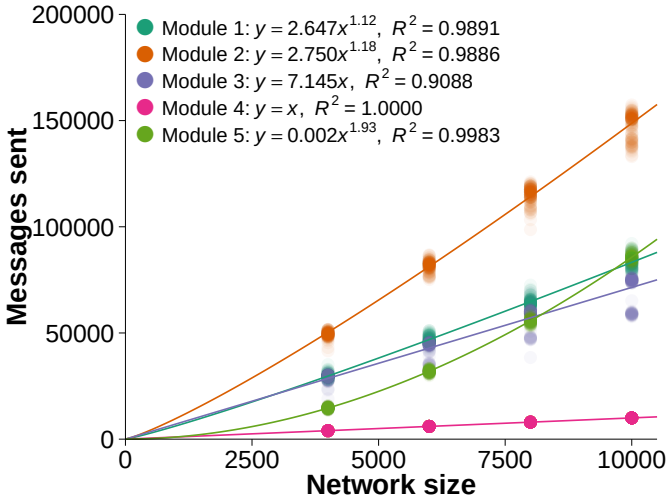



Figure 36: Scalability of communication in terms of the total number of messages sent with constant communication distance for the basicStatic algorithm.

It was also expected that module 1 scaled as a polynomial function (i.e., $O(n^k)$) as the worst case scenario for the number of messages sent for each region component can be calculated as $\sum_{i=0}^{d}(n-i)$, where $n$ is the number of nodes in the sub-network covering the region component and $d$ is the sub-network's diameter.

The number of messages sent by module 5 was expected to scale linearly as it is dependent on the number of nodes changing regions, which increases with network size. This was also the case for module 2, which is a type of surprise flooding algorithm. However, module 5 additionally involves requesting information from neighbors while module 2 involves rebroadcasting messages from neighbors. As the number of neighbors increases with network size due to increasing node density, the combination of these factors results in polynomial instead of linear growth.

In order to potentially reduce the communication complexity of these modules due to increasing node neighborhood sizes, the next experiment reduced the communication distance in proportion to network size. This was done using the formula $\frac{10}{\sqrt{\frac{n}{4000}}}$ where 10 is the initial communication distance and 4000 is the smallest network size. Originally, the average communication neighborhood was approximately 30 nodes at a network size of 4,000 nodes, increasing to approximately 75 at a network size of 10,000. By using this formula, the level of node connectivity remains consistent across network sizes.

This reduction in communication distance in proportion to network size has been applied to the basicStatic algorithm in Figure 37. This graph shows a reduction to sub-polynomial communication complexity for module 2 and a reduction to linear communication complexity for module 5, leaving modules 1, 3, and 4 unchanged from Figure 36.

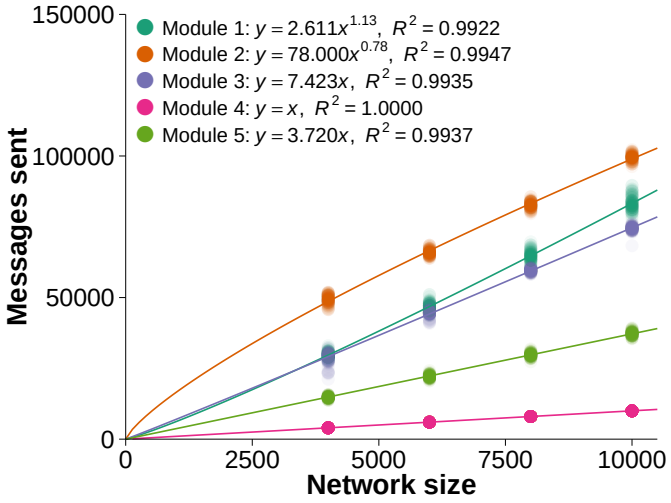



Figure 37: Scalability of communication in terms of the total number of messages sent with proportional communication distance for the basicStatic algorithm.

This was expected as the node neighborhood remained constant in this experiment. In this graph, all modules achieved a better fit than Figure 36, with $R^2$ values of greater than 0.99 observed for all modules. Module 3's increase in fit indicates that a consistent neighborhood size substantially reduced the variation between experimental runs.

In order to further reduce the amount of communication necessary to run the algorithm, message aggregation was implemented in Figure 38. This aggregation was done by having the nodes temporarily store all messages received during a tick, removing any duplicate messages and then processing the messages in a batch. For example, in module 5 of the basicStatic algorithm, nodes changing regions broadcast a request message to their neighbors, which would then send a response. Assuming several of a node's neighbors change regions, this would require that several of the same response message would be sent. By first aggregating these messages, only one response message would need to be sent.

This aggregation of messages can be taken further by comparing the set of messages a node receives at each tick. Consider module 1, in which a node changes its region id every time a lower id is received from its neighbors before rebroadcasting that message. When processing messages individually, a node receiving a sequence of messages with progressively lower ids would change its region id several times and broadcast several messages. By processing a set of messages, only the message with the lowest region id is processed, with the others discarded. The same type of aggregation can be used with module 2, meaning that fewer messages from both modules 1 and 2 will be sent. This reduction in messages sent for modules 1, 2, and 5 can be

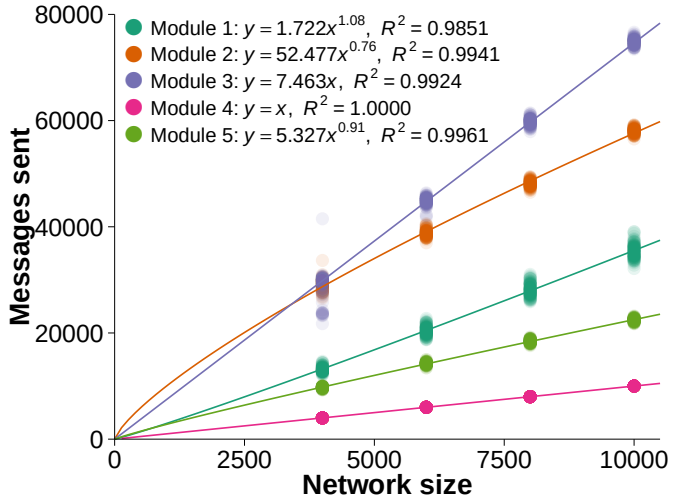



Figure 38: Scalability of communication in terms of the total number of messages sent with proportional communication distance and message aggregation for the basicStatic algorithm.

observed in Figure 38. It is important to note that while message aggregation produces a reduction in the number of messages sent, it has also reduced the order of module 5's communication complexity to sub-polynomial.

Dividing by the number of nodes, each node sent an average of 3.3 and 3.6 messages at network sizes of 4,000 and 10,000 respectively for module 1. This indicates that it took on average of three to four messages for module 1 to select a unique identifier for each region.

For module 2, dividing by the number of nodes an average of 6.3 and 7.8 messages were sent at network sizes of 10,000 and 4,000 respectively. This was expected as the single negative region component has nine adjacent region components, meaning that at least nine messages must be sent by each node in the negative region component. As the positive region components have only a single adjacent region component, nodes in these components send only one hop message. This would lead to an average of approximately five messages being sent. Given that nodes changing regions may also send a message, it is reasonable to assume that an average of approximately six to eight messages are sent by each node per broadcast round.

From this point onward, all further algorithms were tested using communication distances in proportion to network size in addition to message aggregation.

Figure 39 shows the load balance of the basicStatic algorithm on a simple region configuration. From this graph, it can be seen from the fitted lines that while modules 1, 3, 4, and 5 exhibit approximately constant load balance (i.e., the greatest amount of messages sent for any node remains the same regardless of network size), module 2 exhibits negative linear fit, sending at most an average of between 17.5

and 21.6 messages per broadcast round for network sizes of 10,000 and 4,000 respectively. This negative linear fit indicates that the greatest amount of messages sent for any node decreases with increasing network size. This is consistent with module 2's sub-polynomial communication complexity.

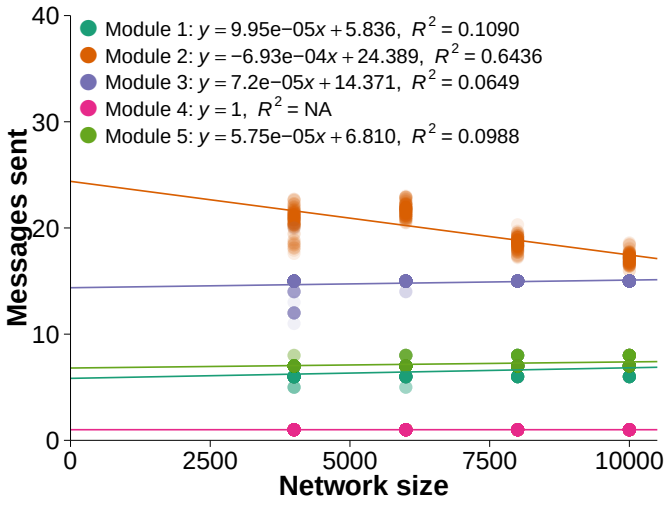



Figure 39: Scalability of communication in terms of worst case (maximum) load (number of messages sent) for any node for the basicStatic algorithm.

For module 1, the average maximum number of messages sent was between 6.2 and 6.8 messages at network sizes of 4,000 and 10,000 respectively. This indicates that the maximum diameter of the largest region component (i.e., the single negative region component) was approximately seven. The average maximum amount of messages sent for module 3 was between 14.7 and 15.1 messages at network sizes of 4,000 and 10,000 respectively. This was consistent with the 14 Voronoi region boundaries detected within the single negative region component. The extra 15th message was likely due to a node changing regions while module 3 was running.

Module 4 sent at most one message regardless of network size, which was expected as module 4 requires that each node send exactly one message. For module 5, the average maximum number of messages sent was between 7 and 7.4 messages at network sizes of 4,000 and 10,000 respectively. This was expected as nodes broadcast messages for module 5 when they change regions as well as when they receive messages from nodes that have changed regions. Given that nodes move seven times during the running of the algorithm, it is likely that a node has changed regions at least once and was adjacent to regions that have changed during the seven movement rounds.

*simpleStatic algorithm*

The next algorithm tested for scalability on a simple region configuration was the simpleStatic algorithm. Figure 40 shows the communication complexity of that algorithm's modules. All regression curves for each of the modules achieved a good fit, as evidenced by $R^2$ values of greater than 0.98. It is important to note that modules 4 and 5 of the simpleStatic algorithm do not broadcast any messages and so were not included in the graph. Given that modules 1 and 2 are identical for every algorithm, it is only module 3 that is of interest. From the graph, it was inferred that module 3 scales linearly in terms of communication complexity, sending approximately 21 messages per node. This was expected for the simple region configuration as messages must be sent for the 14 unique Voronoi region boundaries and seven unique Voronoi junctions.

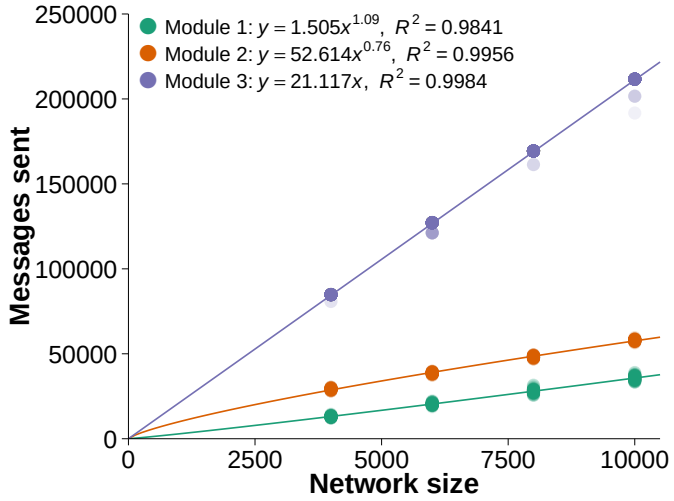



Figure 40: Scalability of communication in terms of the total number of messages sent with proportional communication distance and message aggregation for the simpleStatic algorithm.

Figure 41 shows the load balance of the simpleStatic algorithm. From the fitted lines it can be seen that module 3 exhibits approximately constant load balance, with nodes sending at most approximately 23 messages. The discrepancy between this and an expected maximum of 21 messages can be explained by nodes on a Voronoi junction receiving boundary and junction found messages before detecting the Voronoi junction themselves. This was due to the scheduling system within NetLogo, which processes the algorithm sequentially on each node.

*complexStatic algorithm*

The final algorithm tested for scalability on a simple region configuration was the complexStatic algorithm. Figure 42 shows the communi-

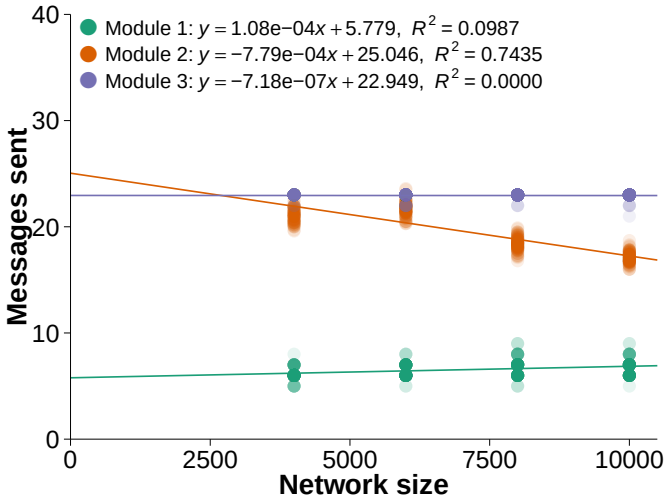



Figure 41: Scalability of communication in terms of worst case (maximum) load (number of messages sent) for any node for the simpleStatic algorithm.

cation complexity of that algorithm's modules. All regression curves for each of the modules achieved a good fit, as evidenced by $R^2$ values of greater than 0.98. Given that modules 1 and 2 are identical for every algorithm, only modules 3–5 are of interest.

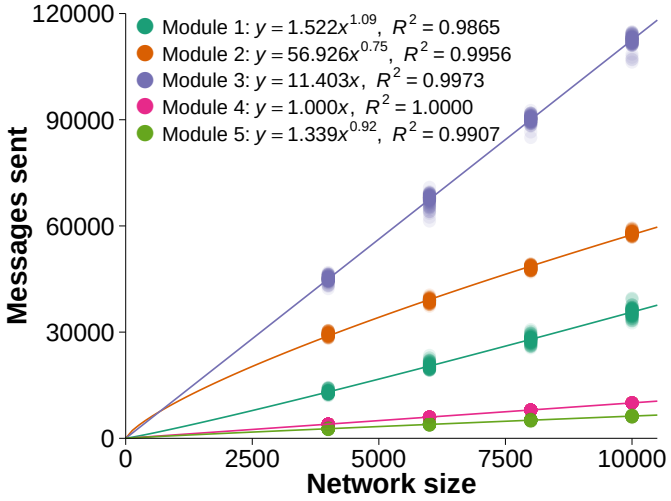



Figure 42: Scalability of communication in terms of the total number of messages sent with proportional communication distance and message aggregation for the complexStatic algorithm.

From the graph, it was inferred that module 3 scales linearly in terms of communication complexity, sending approximately 11.4 messages per node. This was expected for the simple region configuration as the 21 unique Voronoi region boundaries and Voronoi junctions are only broadcast throughout the single negative region component, which will contain approximately half of a network's nodes.

Recall from module 4's code that all nodes within a network must broadcast a maptree message consisting of the portion of the simplified maptree table and label table that is detected within their current region component. This will not be done if that region component detects no Voronoi region boundaries (i.e., nodes within the positive region components). Additionally, nodes receiving a maptree message containing new information will store and rebroadcast that information; therefore the nodes in the positive region components will also send a maptree message, meaning that exactly one message is sent for every node in the network.

From the graph, it was inferred that module 5 is of sub-polynomial order. Module 5 only sends request messages when a node enters a new region if module 4 has not yet run. In this case, an average of between 0.6 and 0.7 messages per node were sent at network sizes of 10,000 and 4,000 respectively. Given that nodes will only respond to these request messages if they are in the same region and have already stored some simplified maptree entries, only nodes entering the negative region component will receive responses. While this would typically result in linear order growth, message aggregation (from proportionally greater nodes being within communication range of multiple nodes sending request messages) has caused sub-polynomial growth.

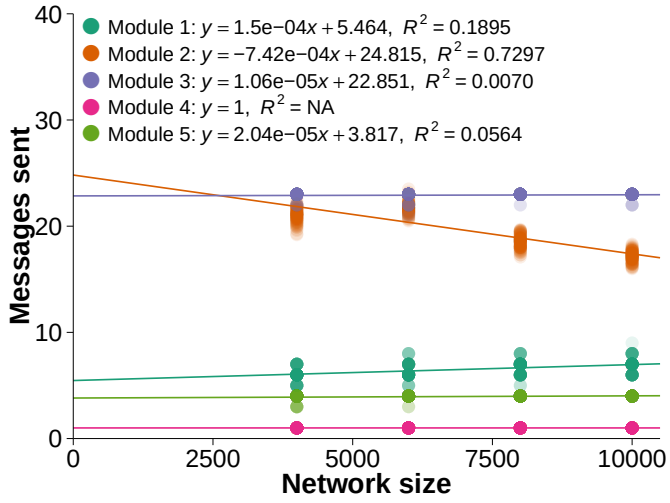



Figure 43: Scalability of communication in terms of worst case (maximum) load (number of messages sent) for any node for the complexStatic algorithm.

Figure 43 shows the load balance of the complexStatic algorithm, with modules 3–5 exhibiting constant load balance. From the fitted lines it can be seen that, like the simpleStatic algorithm, module 3 sends at most approximately 23 messages regardless of network size. Module 4 sends precisely one message per node, which was expected given that only one region (the single negative region component)

produces maptree records. Module 5 sends at most approximately four messages regardless of network size. Given that nodes only send request messages before module 4 has run, and that nodes can move at most four times before module 4 has run, it was expected that nodes are capable of sending at most four messages.

### 6.2.2 *Complex region configuration*

Looking at the complex region configuration (Figure 25), approximately 51% of the area is covered by one of the seven positive region components and 49% is covered by one of the three negative region components. This region configuration consists of 11 unique Voronoi region boundaries, with nine of these boundaries located within the negative region components and two located within the positive region components. This region configuration additionally consists of three unique Voronoi junctions, all of which are located within a negative region component.

#### *basicStatic algorithm*

Figure 44 shows the communication complexity of the basicStatic algorithm running on a complex region configuration. From this graph, it can be inferred that modules 3–5 scaled linearly in terms of communication complexity, whereas modules 1 and 2 were of polynomial and sub-polynomial order respectively. It is important to note that while the number of messages sent differs between that of the basicStatic algorithm running on a simple region configuration, the orders of the modules are unchanged, indicating that region configuration does not affect scalability of the basicStatic algorithm.

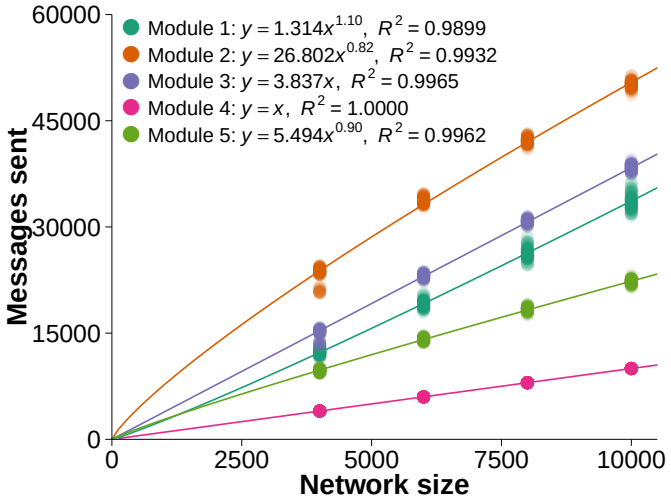



Figure 44: Scalability of communication in terms of the total number of messages sent with proportional communication distance and message aggregation for the basicStatic algorithm.

Module 1 sent overall fewer messages for the complex region configuration than the simple region configuration. Dividing by the number of nodes, each node sent an average of 3 and 3.3 messages at network sizes of 4,000 and 10,000 respectively, as opposed to 3.3 and 3.6 for the simple region configuration. This reduction was due to the complex region configuration having region components with overall smaller sizes. Recall that the simple region configuration has a single negative region component whereas the complex region configuration has multiple negative region components. This reduction in region component size resulted in a corresponding reduction in the diameter of the sub-network that covers the region component.

Module 2 also sent overall fewer messages for the simple region configuration. Dividing by the number of nodes, an average of 5.1 and 6 messages were sent per broadcast round at network sizes of 10,000 and 4,000 respectively, as opposed to the simple region configuration's 6.3 and 7.8 messages. This decrease was due to the complex region configuration having a greater number of adjacent region components (with the complex region configuration having nine adjacent region components and the simple region configuration having only eight), but spread across four region components. Specifically, the largest negative region component has six adjacent region components, three other region components have two, and the remaining six have a single adjacent region component. This is in contrast to the simple region configuration's single negative region component having nine adjacent region components, with the remaining nine region components having a single adjacent region component. Given that nodes changing regions may also send a message, it is reasonable to assume that an average of approximately five to six messages are sent by each node per broadcast round.

Like the previous modules, module 3 sent overall fewer messages, sending approximately 3.8 messages per node as opposed to the simple region configuration's 7.5 messages per node. This was due to the complex region configuration having overall fewer Voronoi region boundaries (11 as opposed to the simple region configuration's 14), and these messages being restricted to overall smaller region components. Specifically, the largest negative region component, (which takes up 44% of the region) contains eight Voronoi boundaries while three other region components, which take up a combined 19% of the region, contain one Voronoi boundary each. Given that the remaining region components contain no Voronoi boundaries, it can be calculated that approximately 3.5 messages should be sent per node. This number assumes a perfectly even distribution of nodes, so an average of 3.8 messages is reasonable.

Modules 4 and 5 remain unchanged from the simple region configuration with the same constant factors. For module 4, this was due to each node sending a single message regardless of region con-

figuration. For module 5, this was due to approximately the same proportion of nodes changing region components.

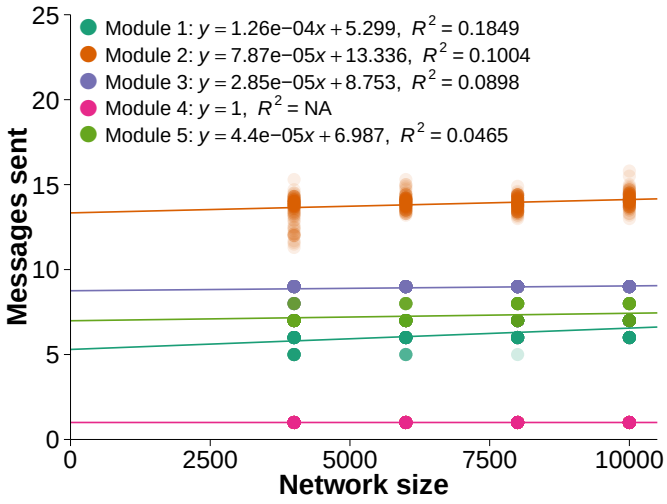



Figure 45: Scalability of communication in terms of worst case (maximum) load (number of messages sent) for any node for the basicStatic algorithm.

Figure 45 shows the load balance of the basicStatic algorithm on a complex region configuration. From this graph, it can be seen from the fitted lines that all modules exhibit an approximately constant load balance (i.e., the greatest amount of messages sent for any node remains the same regardless of network size), which is in contrast to module 2 for the simple region configuration, which exhibited negative linear fit. This change in load balance for module 2 was due to the complex region configuration having a greater number of adjacent region components but overall smaller region component sizes.

For module 1, the average maximum number of messages sent was between 5.8 and 6.6 messages at network sizes of 4,000 and 10,000 respectively. This was in contrast to the 3.3 and 3.6 messages sent for the simple region configuration, indicating that the maximum diameter of the largest region component (i.e., the single negative region component) was smaller, with a size of approximately four as opposed to the simple region configuration's seven.

The average maximum amount of messages sent for module 3 was approximately nine messages, which was consistent with the eight Voronoi region boundaries detected within the largest negative region component. The extra ninth message was likely due to a node changing regions while module 3 was running. Module 4 sent at most one message regardless of network size, which was again expected as module 4 requires that each node send exactly one message.

For module 5, the average maximum number of messages sent was between 7.2 and 7.4 messages at network sizes of 4,000 and 10,000 respectively. This was almost identical to the simple region configura-

tion, again indicating that approximately the same number of nodes changed region components.

*complexStatic algorithm*

The next algorithm tested for scalability on a complex region configuration was the complexStatic algorithm. Figure 46 shows the communication complexity of that algorithm's modules. All regression curves for each of the modules achieved a good fit, as evidenced by $R^2$ values of greater than 0.98. Given that modules 1 and 2 are identical for every algorithm, it is only modules 3–5 that are of interest.

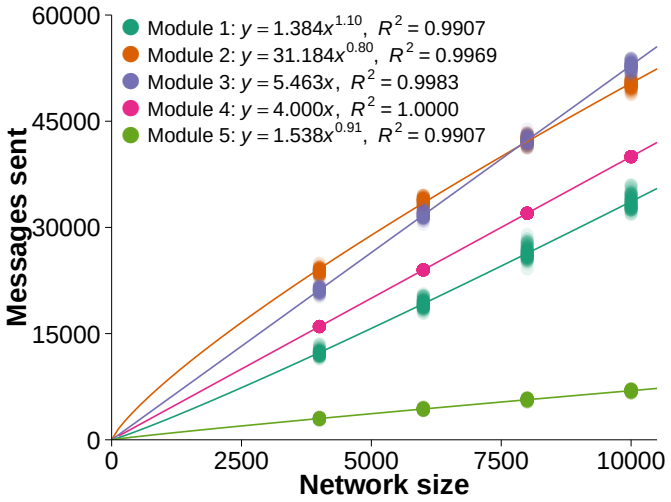



Figure 46: Scalability of communication in terms of the total number of messages sent with proportional communication distance and message aggregation for complexStatic algorithm.

From the graph, it was again inferred that module 3 scaled linearly in terms of communication complexity, sending approximately 5.5 messages per node. This was expected as each Voronoi region boundary and Voronoi junction is only broadcast throughout the region component they are contained within. Specifically, the largest negative region component (which takes up 44% of the region) contains 11 Voronoi boundaries and junctions, while three other region components (which take up a combined 19% of the region) contain only one Voronoi boundary each. Given that the remaining region components contain no region boundaries, it can be calculated that approximately 4.8 messages should be sent per node. As nodes on a Voronoi boundary will additionally broadcast a junction request message, an average of 5.5 messages is reasonable.

For module 4, all nodes within a network must broadcast a maptree message consisting of the portion of the simplified maptree table and label table that is detected within their current region component. This will not be done if that region component detects no Voronoi region boundaries. Additionally, nodes receiving a maptree

message containing new information will store and rebroadcast that information. Given that nodes within four of the region components will have detected unique simplified maptree components, each node must send exactly four messages.

Module 5 remained unchanged from the simple region configuration, due to approximately the same proportion of nodes changing region components.

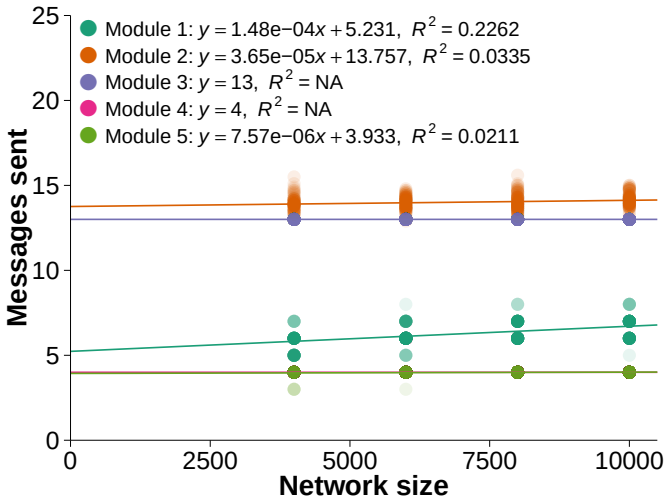



Figure 47: Scalability of communication in terms of worst case (maximum) load (number of messages sent) for any node for the complexStatic algorithm.

Figure 47 shows the load balance of the complexStatic algorithm on the complex region configuration. From the fitted lines it can be seen that module 3 sends at most 13 messages regardless of network size. Like the simpleStatic algorithm, the discrepancy between this and an expected maximum of 11 messages can be explained by nodes on a Voronoi junction receiving boundary and junction found messages before detecting the Voronoi junction themselves due to the scheduling system within NetLogo.

Module 4 sends precisely four message per node, which was expected given that only four region components produce unique maptree records. Like the simple region configuration, module 5 sends at most approximately 4 messages regardless of network size. Again, given that nodes may only send request messages before module 4 has run, and that nodes can move at most four times before module 4 has run, it was expected that nodes are capable of sending at most 4 messages.

## 6.3 SCALABILITY OF DYNAMIC REGIONS

The algorithms tested in this section ran on either the simple region configuration of Figure 55 or the complex region configuration of

Figure 56. In either case, this produced a simulation size of 40×32 patches. The communication distance of the nodes was reduced in proportion to the size of the network, resulting in a communication distance of 3 patch widths for networks with 2,000 nodes and 1.5 for networks with 8,000 nodes. This gave nodes an average communication neighborhood of approximately 44 nodes.

Both the simple and complex regions changed their internal configuration every 100 ticks. It is important to note that during these changes, the boundaries of the region components moved at most a distance of one patch width. Recall from the description of the simpleDynamic algorithm's module 5 (subsection 5.1.6) that changes to the boundaries of the underlying region components should not spontaneously exceed the communication distance of the nodes due to the potential detection of erroneous appearance and disappearance events. Given that the minimum communication distance is 1.5 patch widths, this type of error could not occur.

Node movement occurred every 50 ticks with a movement distance of 0.5 patches, starting at 115 ticks. The broadcast timer was set to 25, starting at 100 ticks. This meant that module 2 was rerun twice as often as the nodes moved. The region timer was set to 100, giving 100 ticks for module 1 to initialize the network. The regionChange timer was reset after five ticks, meaning that any node that had changed regions within the last five ticks was considered to have recently changed its region. Recall that all other timers are reset for the dynamic algorithms, allowing for the algorithms to detect changes to the structure of the underlying region. These timers were all reset to 100 ticks, meaning that the algorithms were capable of detecting any salient changes that occur over a period of 100 ticks or more.

The topologyChange timer was initially set to 110, with the split timer set for 25 ticks later at 135. This gave 25 ticks for the algorithms to detect if a region component had split. The adjacency timer was initially set to 160, giving 25 ticks for any split region components to assign themselves new region component ids. The maptree timer was set for 25 ticks later (at 185), giving 25 ticks for module 3 to complete before module 4 began.

Upon completion, the total number of messages sent by each module for the entire network was recorded (i.e., communication complexity) and the maximum number of messages sent by an individual node for each module (i.e., load balance) was recorded. This was done 100 times with randomized node positions for each network size (2,000, 4,000, 6,000, and 8,000 nodes), leading to a total of 400 experimental runs for each tested algorithm and region configuration.

### 6.3.1 *Simple region configuration*

Looking at the simple region configuration of Figure 55, between 40% and 45% of the area is covered by between three and four positive region components, with the remainder covered by a single negative region component. The following topological events are expected to be detected by the algorithms;

- Split in the large region component between 200 and 300 ticks,
- Unenclose transition for the small region component between 400 and 500 ticks,
- Small region component disappears at 800 ticks,
- Small region component appears at 900 ticks,
- Enclose transition for the small region component between 1,200 and 1,300 ticks, and
- Merge of the large and medium sized region components between 1,400 and 1,500 ticks.

Note that time ranges have been provided for all events bar the appearance and disappearance of regions, which have been given precise times. This is due to differences in communication distance for varying network sizes resulting in differing network granularity. For example, while two region components have clearly split at 200 ticks, the gap between the regions is only two patches wide. Given that the smallest network has a communication distance of three patch widths, these two region components may have nodes that are in direct communication and therefore consider the parts to be a single component. The largest network however has a communication distance of 1.5 patch widths, meaning that the split region components have no direct communication and will therefore detect the split event. Tables 17 and 18 show example dynamic simplified maptree and change tables that would be produced by the simpleDynamic algorithm. In order to detect all changes in the region's configuration, algorithms were run for 1,700 ticks.

*simpleDynamic algorithm*

The first algorithm tested for scalability on a simple region configuration was the simpleDynamic algorithm. Figure 48 shows the communication complexity of that algorithm's modules. All regression curves for each of the modules achieved a good fit, as evidenced by $R^2$ values of greater than 0.99. It is important to note that in the dynamic algorithms, module 4 does not broadcast any messages, and so was not included in any of the graphs of this section.

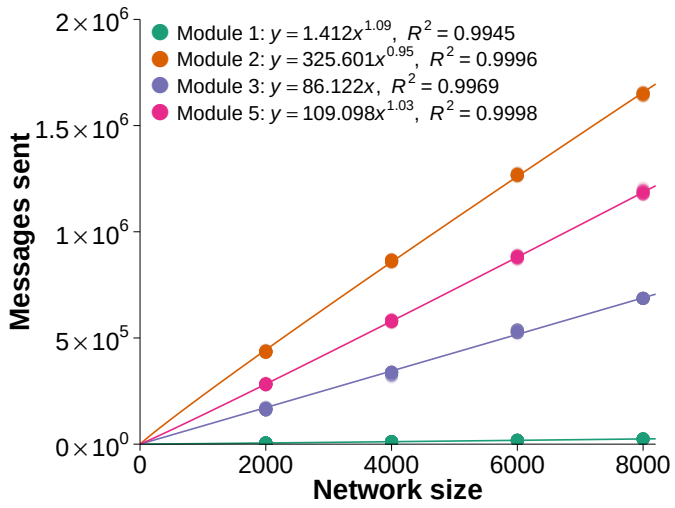



Figure 48: Scalability of communication in terms of the total number of messages sent for the simpleDynamic algorithm.

Given that modules 1 and 2 run identically for every algorithm, respective polynomial and sub-polynomial orders of module 1 and 2's are expected. Dividing by the number of nodes, each node sent an average of 2.8 and 3.2 messages at network sizes of 2,000 and 8,000 respectively for module 1. This indicates that it took an average of two to three messages for module 1 to select a unique identifier for each region.

Dividing by the 68 broadcast rounds and number of nodes, each node sent an average of 3.1 and 3.3 messages at network sizes of 8,000 and 2,000 respectively for module 2. This was expected as the single negative region component has between three and four adjacent region components, meaning that at least three to four hop messages must be sent by each node in the negative region component. As the positive region components have only a single adjacent region component, nodes in those components may send only one hop message. This would lead to an average of approximately 2.3 messages being sent. Given that nodes changing regions may also send a message, it is reasonable to assume that an average of approximately three messages are sent by each node per broadcast round.

From the graph, it was inferred that module 3 scales linearly in terms of communication complexity, sending approximately 86 messages per node. This was expected for the simple region configuration as for the 17 times module 3 is run, there are a total of between 62 and 68 Voronoi region boundaries and between 18 and 22 Voronoi junctions detected. Like the simpleStatic algorithm, these Voronoi region boundaries and Voronoi junctions are broadcast throughout the entire network, leading to a total of between 80 and 90 messages sent per node for module 3. Where a network falls within this range depends on network granularity, with smaller network sizes having

smaller communication distances and therefore being able to detect topological changes as soon as they occur.

It was also expected that module 5 was of polynomial order as it detects the splitting of region components by calculating split ids (*sid*) for each region component. Like module 1, this is done by using leader election. Recall that the worst case scenario for the number of messages sent for each region component can be calculated as $\sum_{i=0}^{d}(n-i)$ where $n$ is the number of nodes in the sub-network covering the region component and $d$ is the sub-network's diameter.

Dividing by the 17 times module 5 is run and by number of nodes, each node sent an average of 8.1 and 8.4 messages at network sizes of 2,000 and 8,000 respectively. Considering that the split detection runs identically to module 1, it is reasonable to assume that approximately three of these messages are due to split detection. Module 5 also has each node broadcast a merge detection message, bringing the total accountable messages up to four. Module 5 also broadcasts a message throughout the network every time a region appearance, split, or merge is detected. Considering that this happens three times during the 17 times module 5 is run, this brings the total number of messages accounted for up to 5.2. The remainder of the messages can be accounted for by nodes sending request and response messages when a node enters a new region.

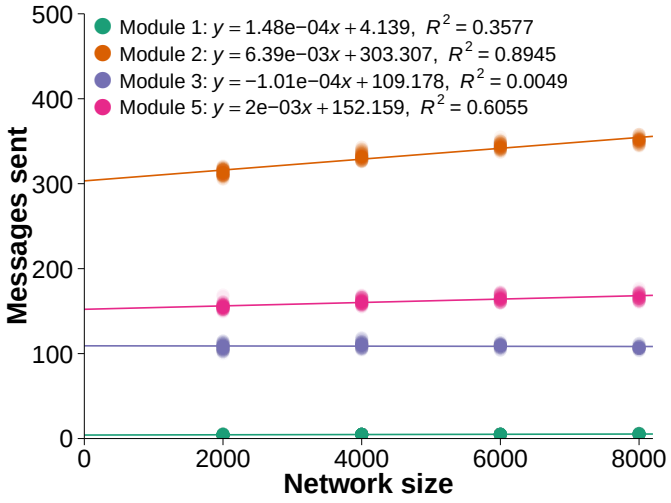



Figure 49: Scalability of communication in terms of worst case (maximum) load (number of messages sent) for any node for the simpleDynamic algorithm.

Figure 49 shows the load balance of the simpleDynamic algorithm on the simple region configuration. From the fitted lines it can be seen that while module 3 exhibits constant load balance (i.e., the greatest amount of messages sent for any node remains the same regardless of network size), modules 1, 2, and 5 exhibit a slightly positive linear fit. This was expected for modules 1 and 5 as their communica-

tion complexity is of polynomial order. For module 2, dividing by the 68 broadcast rounds produces the average maximum number of messages sent by any node in the network per broadcast round. This number of messages was 4.6 for network sizes of 2,000 and 5.2 for network sizes of 8,000. This load balance is consistent with an average number of approximately three messages being sent.

For module 1, the average maximum number of messages sent was between 4.4 and 5.3 messages at network sizes of 2,000 and 8,000 respectively. This indicates that the maximum diameter of the largest region component (i.e., the single negative region component) was approximately five. Dividing by the 17 times module 3 is run, the average maximum number of messages sent by any node was 6.4. This load balance was also contestant with module 3's communication complexity, which showed an average number of approximately five messages being sent each time module 3 was run. For module 5, the average maximum number of messages sent was between 9.2 and 9.9 messages at network sizes of 2,000 and 8,000 respectively. This was again consistent with module 5's communication complexity of approximately eight messages.

*complexDynamic algorithm*

The second algorithm tested for scalability on a simple region configuration was the complexDynamic algorithm. Figure 50 shows the communication complexity of that algorithm's modules. Looking at the graph, it can be seen that the regression curves for each of the modules are almost identical to that of the simpleDynamic algorithm. This is because modules 1, 2, and 5 are identical and module 4 sends no messages. The key difference between module 3 for the simpleDynamic and complexDynamic algorithms is that the complexDynamic's module 3 encloses a region component id and a sensed value in its boundary message. As this does not affect the amount of messages sent, both module 3s broadcast the same number of messages.

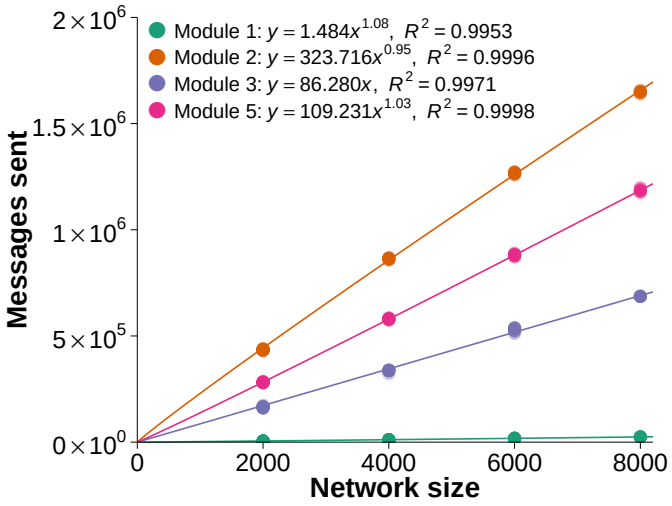



Figure 50: Scalability of communication in terms of the total number of messages sent for the complexDynamic algorithm.

Figure 51 shows the load balance of the complexDynamic algorithm on the simple region configuration. Like Figure 50, the regression curves for each of the modules are almost identical to that of the simpleStatic algorithm.

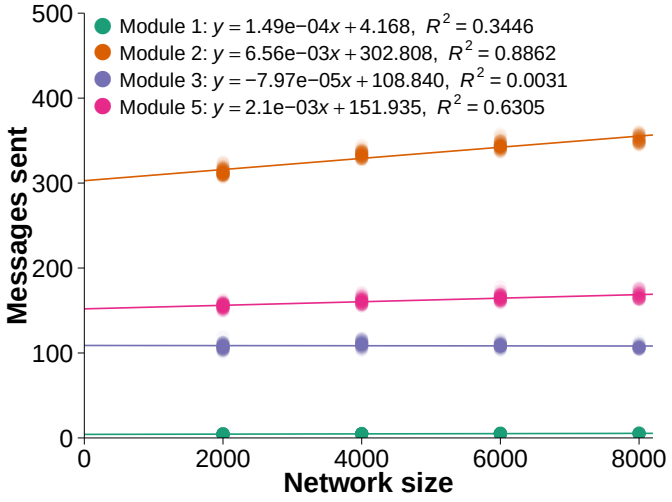



Figure 51: Scalability of communication in terms of worst case (maximum) load (number of messages sent) for any node for the complexDynamic algorithm.

### 6.3.2 *Complex region configuration*

Looking at the simple region configuration of Figure 56, between 40% and 46% of the area is covered by between 3 and 4 positive region components, with the remainder covered by one or two negative region components. The following topological events are expected to

be detected by the algorithms (with only positive region components listed);

- Self-split of the large region component and release transition for the small region component between 200 and 400 ticks,
- Split in the large region component between 500 and 600 ticks,
- Unenclose transition for the small region component between 900 and 1000 ticks,
- Small region component disappears at 1300 ticks,
- Small region component appears at 1400 ticks,
- Enclose transition for the small region component between 1700 and 1800 ticks,
- Merge of the large and medium sized region components between 2100 and 2200 ticks, and
- Self-merge of the large region component and confine transition for the small region component between 2300 and 2500 ticks.

Again note that time ranges have been provided for some events due to differences in communication distance for varying network sizes resulting in differing network granularity. Tables 19, 20, and 21 show example dynamic simplified maptree, change and label tables that would be produced by the complexDynamic algorithm. In order to detect all changes in the complex region's configuration, the algorithm was run for 2,600 ticks.

*complexDynamic algorithm*

Given that the only algorithm capable of successfully detecting changes to a complex region configuration was the complexDynamic algorithm, this will be the only algorithm tested in this section. Figure 52 shows the communication complexity of that algorithm's modules. All regression curves for each of the modules achieved a good fit, achieving $R^2$ values of greater than 0.99. Like the complexStatic algorithm, while the number of messages sent differs between that of the complexDynamic algorithm running on the simple and dynamic region configurations, the orders of the modules are unchanged. This indicates that region configuration and complexity do not affect the scalability of the algorithm.

Module 1 sent overall fewer messages for the complex region configuration than the simple region configuration. Dividing by the number of nodes, each node sent an average of 2.6 and 3 messages at network sizes of 2,000 and 8,000 respectively, as opposed to 2.8 and 3.2 for the simple region configuration. This reduction was due to the complex region configuration's initial configuration having region

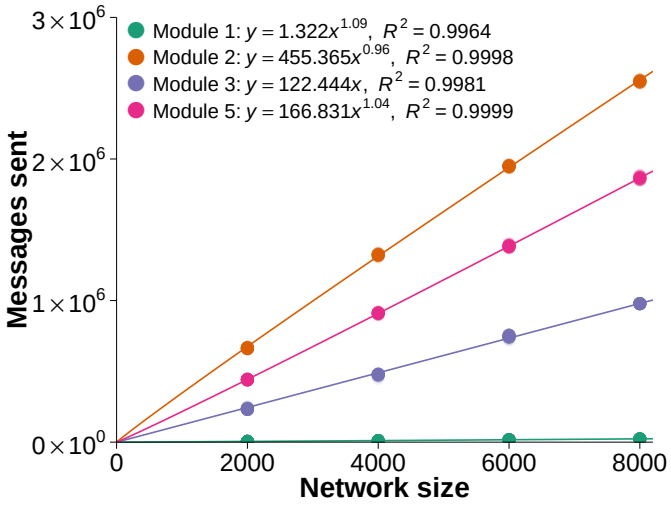



Figure 52: Scalability of communication in terms of the total number of messages sent for the complexDynamic algorithm.

components with overall smaller size. Recall that the simple region configuration has a single negative region component whereas the complex region configuration has two. This slight reduction in region component size resulted in a corresponding reduction in the diameter of the sub-network that covers the region component.

Module 2 also sent approximately the same number of messages for the simple region configuration. Dividing by the 104 broadcast rounds and number of nodes, each node sent an average of 3.1 and 3.2 messages at network sizes of 8,000 and 2,000 respectively as opposed to the simple region configuration's 3.1 and 3.3 messages. This was expected as like the simple region configuration, there is a single negative region component that has between three and four adjacent region components for most of the simulation's runtime.

Dividing by the number of nodes, module 3 sends approximately 122 messages per node. This was expected for the complex region configuration as, for the 26 times module 3 is run, there are a total of between 121 and 133 Voronoi region boundaries junctions detected. Again, where a network falls within this range depends on network granularity, with smaller network sizes having smaller communication distances and therefore being able to detect topological changes as soon as they occur.

For the complex region, module 5 sent an overall greater number of messages when compared to the simple region configuration. Dividing by the 26 times module 5 is run and by number of nodes, each node sent an average of 8.7 and 9.2 messages at network sizes of 2,000 and 8,000 respectively, as opposed to the 8.1 and 8.4 messages sent for the simple region configuration. This increase in the number of messages sent is likely due to the combination of the additional split and merge events over the simple region configuration and more nodes

changing regions when the contains relation is present. More nodes are likely to change regions during the contains relation as that region configuration has a larger proportion of region component boundaries.

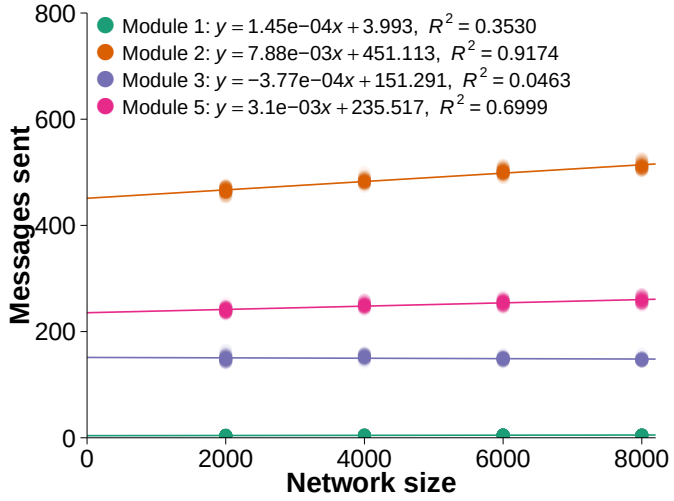



Figure 53: Scalability of communication in terms of worst case (maximum) load (number of messages sent) for any node for the complexDynamic algorithm.

Figure 53 shows the load balance of the complexDynamic algorithm on the complex region configuration. From the fitted lines it can be seen that, like the simple region configuration, while module 3 exhibits constant load balance, modules 1, 2, and 5 exhibit a slightly positive linear fit. This was again expected for modules 1 and 5 as their communication complexity is of polynomial order. For module 2, dividing by the 104 broadcast rounds produces an average maximum number of messages per broadcast round of 4.5 for network sizes of 2,000 and 4.9 for network sizes of 8,000, as opposed to the simple region configuration's 4.6 and 5.2 messages sent. This load balance is again consistent with an average number of approximately three messages being sent.

For module 1, the average maximum number of messages sent was between 4.3 and 5.2 messages at network sizes of 4,000 and 10,000 respectively. This was again slightly smaller than the 4.4 and 5.3 messages sent for the simple region configuration, indicating that the maximum diameter of the largest region component (i.e., the largest negative region component) was slightly smaller.

Dividing by the 26 times module 3 is run, the average maximum number of messages sent by any node was 5.8 in contrast to the simple region configuration's 6.4. This load balance was also contestant with module 3's communication complexity, which showed an average number of approximately five messages being sent each time module 3 was run. For module 5, the average maximum number of

messages sent was between 9.3 and 10 messages at network sizes of 2,000 and 8,000 respectively, which was slightly larger than the simple region configuration's 9.2 and 9.9 messages sent. This was again consistent with module 5's communication complexity of approximately nine messages.

## 6.4 SUMMARY

This chapter has presented experimental evaluations of the five algorithms presented in Chapters 4 and 5. These algorithms have been evaluated in terms of both veracity and scalability.

Section 6.1 evaluated the veracity of modules 1 and 2. As all of the algorithms require correct information from these two modules to construct their specific formal models, meeting the necessary requirements for modules 1 and 2 to perform correctly will produce correct results for the entirety of the algorithms.

Module 1 requires that there is at least one node in each region component, and that all nodes within that region component form a connected sub-graph. Additionally, module 1 requires that the communication distance of the nodes is smaller than the smallest distance between any two positive or any two negative region components. This requirement is to ensure that two separate regions are not falsely counted as one. The results of Figure 34 confirmed these findings.

Module 5 of each algorithm is responsible for swapping a node's region component id and Voronoi region component id when it enters a new region component. For a correct swap, that node must have the correct ids prior to changing regions. For this to occur the Voronoi boundaries must be in the correct location, which for a mobile geosensor network is done by periodically rerunning module 2. How often module 2 must be run can be determined by the formula $b \leq \frac{d}{2s}$ where $d$ is the shortest distance between any two positive or negative region components, $s$ is the maximum speed of the nodes, and $b$ is the broadcast interval of module 2. These findings were confirmed by the experiment shown in Figure 35.

The findings for modules 1 and 2 were used when implementing the scalability experiments of sections 6.2 and 6.3, ensuring that correct results would be obtained. Table 12 shows a summary of the communication complexity of the algorithms tested on static region configurations (i.e., the basicStatic, simpleStatic, and complexStatic algorithms).

From this table, it can be seen that all regression curves for each of the algorithm's modules achieved a good fit, as evidenced by $R^2$ values of greater than 0.98. Looking at the regression curves, only module 1 is of polynomial order, with all other modules being of either linear or sub-polynomial order. This is because module 1's broadcasting characteristics are based on leader election algorithms whereas

| | | Simple region | | Complex region | |
|---|---|---|---|---|---|
| | Module | Regression | $R^2$ | Regression | $R^2$ |
| basicStatic | 1. | $y = 1.722x^{1.08}$ | 0.9851 | $y = 1.314x^{1.10}$ | 0.9899 |
| | 2. | $y = 57.447x^{0.76}$ | 0.9941 | $y = 26.802x^{0.82}$ | 0.9932 |
| | 3. | $y = 7.463x$ | 0.9924 | $y = 3.837x$ | 0.9965 |
| | 4. | $y = 1.000x$ | 1.0000 | $y = 1.000x$ | 1.0000 |
| | 5. | $y = 5.327x^{0.91}$ | 0.9961 | $y = 5.494x^{0.90}$ | 0.9962 |
| simpleStatic | 1. | $y = 1.505x^{1.09}$ | 0.9841 | | |
| | 2. | $y = 52.614x^{0.76}$ | 0.9956 | | |
| | 3. | $y = 21.117x$ | 0.9984 | | |
| | 4. | | | | |
| | 5. | | | | |
| complexStatic | 1. | $y = 1.522x^{1.09}$ | 0.9865 | $y = 1.384x^{1.10}$ | 0.9907 |
| | 2. | $y = 56.926x^{0.75}$ | 0.9956 | $y = 31.184x^{0.80}$ | 0.9969 |
| | 3. | $y = 11.403x$ | 0.9973 | $y = 5.463x$ | 0.9983 |
| | 4. | $y = 1.000x$ | 1.0000 | $y = 4.000x$ | 1.0000 |
| | 5. | $y = 1.339x^{0.92}$ | 0.9907 | $y = 1.538x^{0.91}$ | 0.9907 |

Table 12: Scalability of static algorithms in terms of the total number of messages sent by each module. Regression curve for module 2 shows the number of messages sent per broadcast round.

modules 2–4 are based on surprise flooding and module 5 is based on requests from and responses to nodes that are changing regions.

It is important to note that while the number of messages sent differs between the simple region configuration and the complex region configuration for the same algorithm, the order of the modules is unchanged. This consistency indicates that region configuration does not affect scalability of the static algorithms.

Given that modules 1 and 2 are the same for all algorithms, these two modules can be ignored when comparing the scalability of two algorithms. Starting with the simple region configuration, the basicStatic algorithm sends an average of approximately 11 messages per node whereas the simpleStatic algorithm sends approximately 21 and the complexStatic sends approximately 13. While this would indicate that the basicStatic algorithm is the best for use on a simple region configuration, given that only the simpleStatic and complexStatic algorithms are capable of distinguishing between engulfs and surrounds relations, the complexStatic algorithm is of greater value.

It is interesting to note that the simpleStatic algorithm performed far worse than the complexStatic algorithm, sending almost double the number of messages per node. This is due to the complexStatic al-

gorithm confining its module 3 messages to individual region components. It is clear from this that computing segments of the simplified maptree in individual region components and then broadcasting the completed segments throughout the network has lead to gains in efficiency. This leads to the conclusion that the complexStatic algorithm is the best for any static region configuration.

Table 13 shows a summary of the communication complexity of the algorithms tested on dynamic region configurations (i.e., the simpleDynamic and complexDynamic algorithms).

| | | Simple region | | Complex region | |
|---|---|---|---|---|---|
| | Module | Regression | $R^2$ | Regression | $R^2$ |
| simpleDynamic | 1. | $y = 1.412x^{1.09}$ | 0.9945 | | |
| | 2. | $y = 325.601x^{0.95}$ | 0.9996 | | |
| | 3. | $y = 86.122x$ | 0.9969 | | |
| | 4. | | | | |
| | 5. | $y = 109.098x^{1.03}$ | 0.9998 | | |
| complexDynamic | 1. | $y = 1.484x^{1.08}$ | 0.9953 | $y = 1.322x^{1.09}$ | 0.9964 |
| | 2. | $y = 323.716x^{0.95}$ | 0.9996 | $y = 455.365x^{0.96}$ | 0.9998 |
| | 3. | $y = 86.280x$ | 0.9971 | $y = 122.444x$ | 0.9981 |
| | 4. | | | | |
| | 5. | $y = 109.231x^{1.03}$ | 0.9998 | $y = 166.831x^{1.04}$ | 0.9999 |

Table 13: Scalability of dynamic algorithms in terms of the total number of messages sent by each module.

From this table it can be seen that all regression curves for each of the algorithm's modules achieved a good fit, as evidenced by $R^2$ values of greater than 0.99. Looking at the regression curves, modules 1 and 5 are of polynomial order, with modules 2 and 3 being of subpolynomial and linear order respectively. This is again because the broadcasting characteristics of modules 1 and 5 are at least in part based on leader election algorithms, whereas modules 2 and 3 are based on surprise flooding algorithms.

As was seen in the static regions, while the number of messages sent differs between the simple region configuration and the complex region configuration, the orders of the modules are unchanged. This consistency indicates that region configuration does not affect scalability of the dynamic algorithms.

Looking at the simple region configuration, it can be seen that the regression curves for each of the modules are almost identical for the simpleDynamic and complexDynamic and algorithms. This is because modules 1, 2, and 5 are identical and module 4 sends no messages. Recall that the key difference between module 3 for the simpleDynamic and the complexDynamic algorithms is that the complexDy-

namic's module 3 encloses a region component id and a sensed value in its boundary message. As this does not affect the amount of messages sent, both module 3s broadcast the same number of messages. Given this information, this leads to the conclusion that the complexDynamic algorithm is the best for any dynamic region configuration.

# 7

# CONCLUSIONS

This thesis has demonstrated how regions with complex internal structures can be qualitatively described. Additionally, such regions may be dynamic in nature, in that they can reconfigure their internal structure over time. This research has proposed in-network, decentralized algorithms that qualitatively describe the internal structure of and changes to these regions. Unlike previous work, these algorithms are able to operate in networks of mobile geosensor nodes with no access to coordinate information. These algorithms make use of qualitative spatial reasoning and the simplified maptree formal model to efficiently record only salient changes to the internal structure of these regions.

## 7.1 RESULTS AND MAJOR FINDINGS

This thesis has investigated the monitoring of both static and dynamic region configurations that have a simple or complex internal structure. To do so, a collection of five decentralized algorithms were designed and tested. The major findings of this research are as follows:

### *Algorithm efficiency*

Firstly, the veracity of modules 1 and 2, which were the same for all algorithms, was investigated. This was because if the necessary requirements for modules 1 and 2 to perform correctly were met, the rest of the algorithm would produce correct results. These experiments found that the criteria of sufficient node density, broadcast interval, and a communication distance that is smaller than the minimum distance between any two positive or negative region components would produce correct results. These three factors would be dependent on the characteristics of the phenomena being monitored.

The veracity experiments additionally illustrated the type of tradeoffs that occur when designing decentralized algorithms, specifically between network size and communication distance. For example, a small network size and a large communication distance can produce the same level of accuracy as a large network with a small communication distance. Given that the maximum communication distance is restricted by region configuration, and that transmission costs increase rapidly with communication distance, it would be more cost effective to have a large network with a small communication distance.

For the efficiency of the static algorithms, it was found that the complexStatic algorithm provided the best balance of information detail and algorithm efficiency for static regions, whereas for the dynamic algorithms it was found that both algorithms performed identically. Given that the simpleDynamic algorithm is only capable of functioning on regions with a simple internal configuration, the complexDynamic algorithm is the better choice for dynamic regions.

It was found that all modules of the algorithms exhibited either sub-polynomial, linear, or weakly polynomial scalability (with the worst case being $O(n^{1.1})$). The order of scalability produced was due to the type of decentralized algorithm the module was based on, with leader election based algorithms producing weakly polynomial scalability (e.g., module 1), and surprise flooding algorithms producing sub-polynomial or linear scalability (e.g., modules 2 and 3 respectively for any algorithm).

In addition to algorithm scalability, it is important to consider the costs of periodically running the modules, in particular module 5 of the dynamic algorithms, which exhibits weakly polynomial scalability. Given that the nodes in a dynamic geosensor network have limited battery capacity, such modules can only be run a certain number of times before the battery is depleted. In the interests of extending network run-time, it is therefore important to consider how often particular modules should be run. For the dynamic algorithms, such a trade off would then be between the temporal granularity of the the records and the network's lifespan.

*Qualitative relations*

In order to store the qualitative internal structure of a region, the simplified maptree formal model was used. The simplified maptree was based on the standard maptree [28], which is a black-white edge labeled tree based on combinatorial maps and containment trees capable of uniquely representing the topological structure of regions.

Instead of storing the relations between region components, the simplified maptree stores Voronoi region components that are induced by region components, of which there is a 1:1 mapping. Additionally, the simplified maptree dispenses with unique edge identification and the boundary cycles of region components, instead storing only region component adjacency.

Building on the simplified maptree, the following three qualitative spatial relations were produced:

- *Contains*, where a single region component completely encloses another region component,
- *Engulfs*, where a single region component partially encloses another region component, and

- *Surrounds*, where multiple region components partially enclose another region component.

By extending the simplified maptree to account for region dynamism, the specific way region configurations enter and exit the three qualitative spatial relations was able to be described using a conceptual neighborhood graph. This conceptual neighborhood graph provided a detailed description of how the relations between region components can change over time.

*Evaluation of hypothesis*

Looking back at the hypothesis presented at the start of this thesis (Section 1.2), the individual components can be answered as follows:

Decentralized algorithms have been designed that successfully detect and monitor a variety of complex spatial phenomena. These algorithms have been shown to:

1. Operate successfully without any need for coordinate information,
2. be tolerant of node mobility, successfully running on dynamic geosensor networks,
3. accurately detect the qualitative structure of the underlying region to varying extents; from the basicStatic algorithm being unable to distinguish between the surrounds and engulfs relations, to the complexStatic algorithm that distinguished between the surrounds, engulfs, and contains relations,
4. correctly detect salient changes to the qualitative structure of dynamic regions, with the simpleDynamic and complexDynamic algorithms successfully recording the specifics of how region components enter and exit qualitative relations, and
5. demonstrate from the results of the experiments that all algorithms were able to accurately function while being efficient in the amount of communication needed, with no module exceeding a weak polynomial communication complexity.

## 7.2 LIMITATIONS AND FUTURE WORKS

While the algorithms presented in this thesis have been shown to be both accurate and efficient, there are some limitations that represent avenues for future research.

*Node volatility*

When formally constructing the dynamic geosensor network used by the algorithms, node communication was modeled with the graph $G(t) = (V, E(t))$. While this graph allows for time varying communication (i.e., communication links can form and break over time), it assumes node permanence. This may not always be the case, as some nodes may enter or exit the geosensor network at various times. This may be from nodes deactivating due to energy depletion, leaving the area they are tasked to monitor, or a technical failure leading to a breakdown. Additionally, new nodes may be introduced to replace nodes that have failed. This volatility can be formally modeled with the communication graph $G(t) = (V(t), E(t))$, which describes a volatile dynamic geosensor network.

Given that most field implementations of geosensor networks use nodes with limited battery capacity, this limits the network to a finite lifespan. Assuming that the algorithms described in this work were modified to allow for new nodes to be introduced, this would allow the area of interest to be monitored indefinitely. For the algorithms described in this work, a gradual loss of nodes over time would not introduce errors provided that there is sufficient coverage of the evaluated area. Recall from Section 6.1 that when node coverage becomes too sparse, errors are quick to accumulate. Coverage could be kept up by introducing new nodes to the network at approximately the same rate that nodes drop out. New code could be added so that newly introduced nodes request information from their neighbors. This code would be similar to that of the request and response messages used when a node enters a new region in the complexStatic algorithm.

*Node unreliability*

Common to the formal specifications of all algorithms in this work is that the nodes' communication and sensing capabilities are assumed to be reliable. In field implementations of decentralized algorithms, this has proven to not always be the case, with nodes occasionally failing to receive messages or their sensors returning incorrect data.

The occasional dropped message would be unlikely to affect the accuracy of the algorithms described in this work as they are based on leader election and surprise flooding algorithms. These types of algorithms require that nodes receiving a message pass that message on if certain criteria are met. This means that any node that drops a message would still receive that message, or a similar one, by another of its neighbors.

Sensor inaccuracy would however be a problem for these algorithms. In particular, consider how the dynamic algorithms detect the appearance of a new region component. Recall from section 5.1.6 that nodes detecting a change in their sensed value will request the

sensed values of their neighbors. If all of the received sensed values do not match the node's own sensed value, the node will then assume that a new region component has appeared. If this change in sensed value was due to sensor error, then the detected appearance event would also be in error. In order to eliminate such occurrences, the algorithm's code would need to be altered so that additional nodes are required to confirm the detection of such events.

Both sensor and communication inaccuracy could be simulated with the introduction of both message and sensor probability functions. The probability that a node will send or receive a message as well as the probability that the node senses the correct value could then be altered in order to test the veracity of the algorithms at various levels of node unreliability.

*Dynamic region simulation*

Two main approaches present themselves when simulating dynamic regions; either import a sequence of images, or algorithmically generate a randomized dynamic region within the simulation. The first method was used in this work as the specific topological events were known ahead of time, allowing for verification of results as well as repeatability of the experiments. In addition, data from real-world phenomena such as algal blooms could be imported using this method, which would allow for the effectiveness of these algorithms to be demonstrated for a wide range of phenomena.

Given that the type and frequency of topological changes occurring to a randomized dynamic would be unknown, this unpredictability would allow for a more comprehensive test of the algorithms' veracity. Initial work for simulating dynamic regions has been completed for use in testing some of the algorithms presented in [8], which could be extended to log topological events. This would enable the testing of the algorithm's veracity.

By additionally using data from real-world phenomena as well as randomized dynamic regions, a more comprehensive evaluation of the algorithms would be possible.

*Network synchronization*

In order to keep costs low, most field implementations equip nodes with an oscillating crystal to keep time. Given that the oscillation rate of these crystals is subject to voltage and temperature, over time the clocks of individual nodes will begin to differ. This is known as clock drift.

Common to all of the algorithms presented in this work is the reliance on timers to ensure that the various components of the algorithms are run correctly. For example, module 1 uses the region timer to specify how long the network should wait before assigning a

unique identifier to each region component and then running module 2. This region timer is assumed to expire at the same time for every node, which requires that every node's clock increments by the same amount at the same time. This is a property known as clock synchronization. In addition to coordinating the timers, the nodes' clocks are used in the dynamic algorithms to log when topological events occur.

Substantial work has been carried out developing decentralized algorithms that are capable of synchronizing the clocks of geosensor networks [85]. A simple example of a synchronization protocol employed by many decentralized algorithms is round trip synchronization, where a node sends a message to a neighbor requesting that node's current timestamp [8]. When the node receives a response, it then knows that the received timestamp lies somewhere between when the request was sent and the response received. By repeating this process, a collection of time differences is recorded that can then used to estimate the relative clock drift of the two nodes. This method is fairly expensive in terms of communication complexity, requiring a set of these messages be sent for each communication link in the network. For networks with dense communication graphs, this would lead to polynomial communication complexity. Given that after the synchronization algorithm has run the clocks will again begin to drift apart, the algorithm would need to be run periodically. This would substantially decrease the algorithm's efficiency, reducing network lifespan.

An alternate approach to running clock synchronization algorithms would be to record topological events at coarser granularity and to dispense with the use of timers for the coordination of the algorithms' various components. This could be done by using a mass-based approach to the detection of topological changes, similar to that of the algorithms discussed in section 2.3.3. For example, while the algorithms of chapter 5 are capable of immediately detecting the appearance of new region components; disappearance, split, and merge events are only detected by periodically checking for their occurrence. If each region component were aware of the approximate number of nodes it contained, sudden increases or decreases would be an indicator of merge and split events respectively whereas a gradual change towards zero would be an indicator of a disappearance event. Modules 3 and 4 would then only need to run after these events instead of periodically.

*Increasing algorithm capabilities*

Presently the algorithms displayed in this work are capable of completely describing the topological structure of a region using the simplified maptree. This means that while the relationships between region components are known, nothing is known about the absolute size, shape, or position of these region components. This was inten-

tional as this work has focused on using qualitative spatial reasoning to describe the structure of a region as opposed to a quantitative record of the region's structure as a set of polygons.

It would however be useful to include the relative sizes of the region components. Looking at the algorithms of section 2.3.3, it is possible to compute the relative size of region components using mass-based approaches. If the white nodes of the simplified map-tree (i.e., the nodes representing the region components) were to be additionally labeled with their relative size, it would provide a more comprehensive understanding of the region's structure. Additionally, this would allow the dynamic algorithms of chapter 5 to describe expansion and contraction of region components alongside topological events.

On the subject of the dynamic algorithms, a limitation of these algorithms is that region components can split into at most two components and at most two components can merge. This means that topological events where additional region components merge or split, such as the region configuration shown in Figure 54, are not able to be correctly detected. Such events could however be accommodated by altering the change table ($C_t$), specifically, by replacing the part columns (i.e., $p_1$ and $p_2$) with a single column that stores a set of region components.



Figure 54: Dynamic region example showing the simultaneous merging of three region components.

Additionally, topological events were treated as atomic by the dynamic algorithms, with at most one topological event occurring at a single time step. While the code for module 5 is capable of detecting multiple topological events, for the sake of simplicity the code of module 4, which records these topological changes, is not. It would therefore simply be a case of updating the module 4 code so that the dynamic algorithms are capable of recording these changes.

## 7.3 FINAL THOUGHTS

While this thesis has focused chiefly on the testing of the decentralized algorithms presented in this work on simulated dynamic geosensor networks, the practical implications of this work must also be considered. Although the qualitative spatial relations and simplified

maptrees described in this work are not limited to decentralized algorithms, it is useful to use a geosensor network as an example.

Consider the task of monitoring changes to the structure of dynamic spatial phenomena on the ocean's surface, for instance algal blooms or oil spills. Particularly during storms, such areas would not have sufficient visibility for satellite imagery and would suffer from inadequate location accuracy due to poor GNSS (Global Navigation Satellite System) reception. A network of buoyant sensor nodes could however be scattered across the sea surface, with wind and ocean currents aiding in their dispersal. The algorithms described in this work would then make use of qualitative spatial reasoning to efficiently record only salient changes to the internal structure of the monitored phenomena. These nodes would then require only a single sensor to detect the presence or absence of the phenomena and a wireless transceiver to communicate with neighboring nodes. This, in addition to not requiring a GNSS radio, would reduce both the cost and the power requirements of the network.

This work could be of great benefit in bringing long-term environmental monitoring to areas previously unable to be monitored due to their location, the cost of deploying a suitable geosensor network, or the time required to be monitored.

# A

## APPENDIX

| Halfedge | Twin | Next | Previous | Face | Component |
|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | 0 | -1 |
| a | $\bar{a}$ | b | d | 0 | M1 |
| $\bar{a}$ | a | e | g | 1 | M1 |
| b | $\bar{b}$ | c | a | 0 | M1 |
| $\bar{b}$ | b | $\bar{g}$ | j | 2 | M1 |
| c | $\bar{c}$ | d | b | 0 | M1 |
| $\bar{c}$ | c | $\bar{j}$ | $\bar{j}$ | 3 | M1 |
| d | $\bar{d}$ | a | c | 0 | M1 |
| $\bar{d}$ | d | j | $\bar{e}$ | 2 | M1 |
| e | $\bar{e}$ | f | $\bar{a}$ | 1 | M1 |
| $\bar{e}$ | e | $\bar{d}$ | h | 2 | M1 |
| f | $\bar{f}$ | g | e | 1 | M1 |
| $\bar{f}$ | f | $\bar{h}$ | $\bar{h}$ | 4 | M1 |
| g | $\bar{g}$ | $\bar{a}$ | f | 1 | M1 |
| $\bar{g}$ | g | h | $\bar{b}$ | 2 | M1 |
| h | $\bar{h}$ | $\bar{e}$ | $\bar{g}$ | 2 | M1 |
| $\bar{h}$ | h | $\bar{f}$ | $\bar{f}$ | 4 | M1 |
| i | $\bar{i}$ | i | i | 2 | M3 |
| $\bar{i}$ | i | $\bar{i}$ | $\bar{i}$ | 5 | M3 |
| j | $\bar{j}$ | $\bar{b}$ | $\bar{d}$ | 2 | M1 |
| $\bar{j}$ | j | $\bar{c}$ | $\bar{c}$ | 3 | M1 |
| k | $\bar{k}$ | l | m | 3 | M2 |
| $\bar{k}$ | k | n | o | 6 | M2 |
| l | $\bar{l}$ | m | k | 3 | M2 |
| $\bar{l}$ | l | $\bar{o}$ | p | 7 | M2 |
| m | $\bar{m}$ | k | l | 3 | M2 |
| $\bar{m}$ | m | $\bar{p}$ | $\bar{n}$ | 8 | M2 |
| n | $\bar{n}$ | o | l | 6 | M2 |
| $\bar{n}$ | n | $\bar{m}$ | $\bar{p}$ | 8 | M2 |
| o | $\bar{o}$ | $\bar{k}$ | n | 6 | M2 |
| $\bar{o}$ | o | p | $\bar{l}$ | 7 | M2 |
| p | $\bar{p}$ | $\bar{l}$ | $\bar{o}$ | 7 | M2 |
| $\bar{p}$ | p | $\bar{n}$ | $\bar{m}$ | 8 | M2 |

Table 14: DCEL table augmented with connected component labeling. Complete version of Table 2.

| $rid_a$ | $rid_b$ | $cid$ |
|---|---|---|
| -1 | 0 | -1 |
| 0 | 1 | M1 |
| 0 | 2 | M1 |
| 0 | 3 | M1 |
| 1 | 2 | M1 |
| 1 | 4 | M1 |
| 2 | 3 | M1 |
| 2 | 4 | M1 |
| 2 | 5 | M3 |
| 3 | 6 | M2 |
| 3 | 7 | M2 |
| 3 | 8 | M2 |
| 6 | 7 | M2 |
| 6 | 8 | M2 |
| 7 | 8 | M2 |

Table 15: Simplified maptree table based on Table 14.

| $rid_a$ | $rid_b$ | $cid$ | $neg$ |
|---|---|---|---|
| -1 | 0 | -1 | 0 |
| 0 | 2 | a | 0 |
| 0 | 3 | a | 0 |
| 0 | 4 | a | 0 |
| 1 | 8 | d | 1 |
| 1 | 9 | e | 1 |
| 2 | 3 | a | 0 |
| 2 | 5 | b | 0 |
| 3 | 4 | a | 0 |
| 3 | 6 | c | 0 |
| 3 | 7 | a | 0 |
| 4 | 7 | a | 0 |

Table 16: Simplified maptree table based on Figure 25.

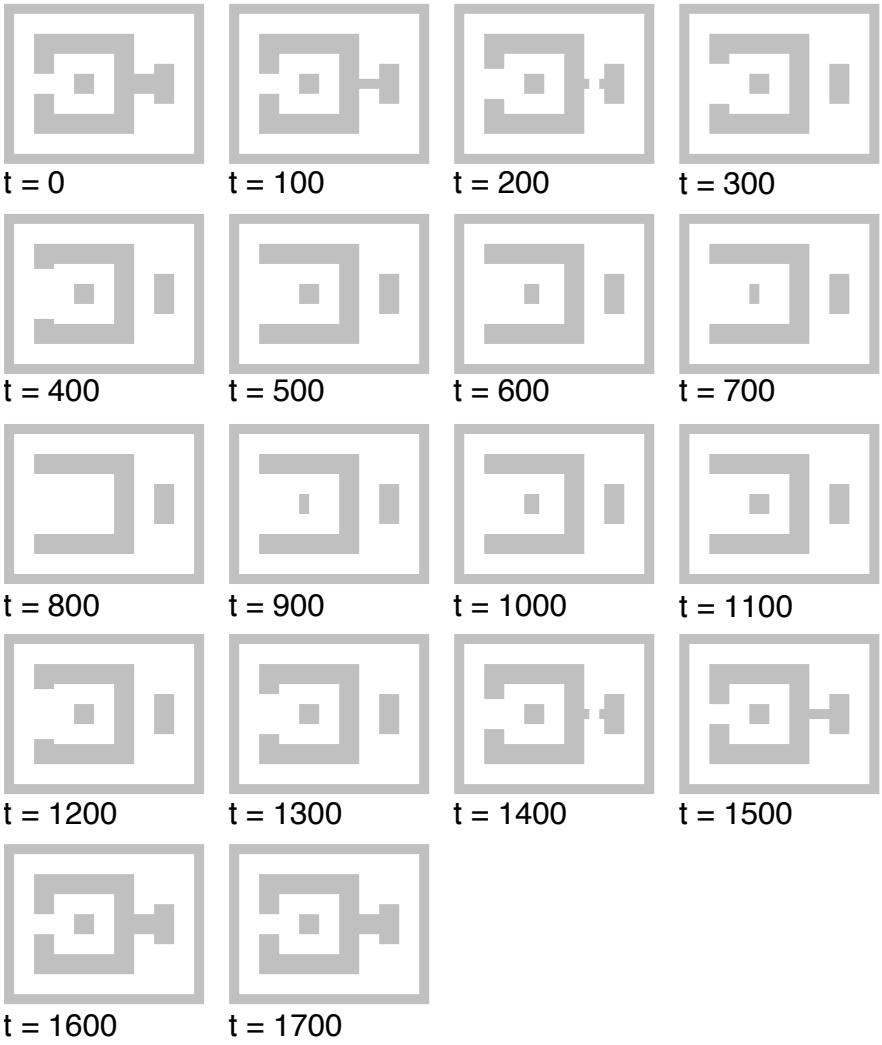



Figure 55: Simple dynamic region used in the evaluation of algorithms from chapter 5. Region dimensions are 40×32 and t indicates the time the region enters the shown configuration.

| $rid_a$ | $rid_b$ | $cid$ | $t_1$ | $t_2$ |
|---|---|---|---|---|
| -1 | 0 | -1 | 0 | ∅ |
| 0 | 1 | a | 100 | 200 |
| 1 | 2 | b | 100 | 200 |
| 0 | 3 | a | 200 | 500 |
| 0 | 4 | a | 200 | 500 |
| 3 | 4 | a | 200 | 500 |
| 2 | 3 | b | 200 | 500 |
| 0 | 3 | c | 500 | 1300 |
| 0 | 4 | c | 500 | 1300 |
| 3 | 4 | c | 500 | 1300 |
| 2 | 3 | c | 500 | 800 |
| 0 | 2 | c | 500 | 800 |
| 3 | 5 | c | 900 | 1300 |
| 0 | 5 | c | 900 | 1300 |
| 0 | 3 | d | 1300 | 1500 |
| 0 | 4 | d | 1300 | 1500 |
| 3 | 4 | d | 1300 | 1500 |
| 3 | 5 | e | 1300 | 1500 |
| 0 | 6 | d | 1500 | ∅ |
| 5 | 6 | e | 1500 | ∅ |

Table 17: Simplified dynamic maptree table based on Figure 55 where split detected at 200, unenclose transition at 500, disappearance at 800, appearance at 900, enclose transition at 1300, and merge detected at 1500.

| $w$ | $p_1$ | $p_2$ | $split$ | $t$ |
|---|---|---|---|---|
| 1 | 3 | 4 | TRUE | 200 |
| c | a | b | FALSE | 500 |
| c | d | e | TRUE | 1300 |
| 6 | 3 | 4 | FALSE | 1500 |

Table 18: Change table based on Figure 55 where split detected at 200, unenclose transition at 500, enclose transition at 1300, and merge detected at 1500.

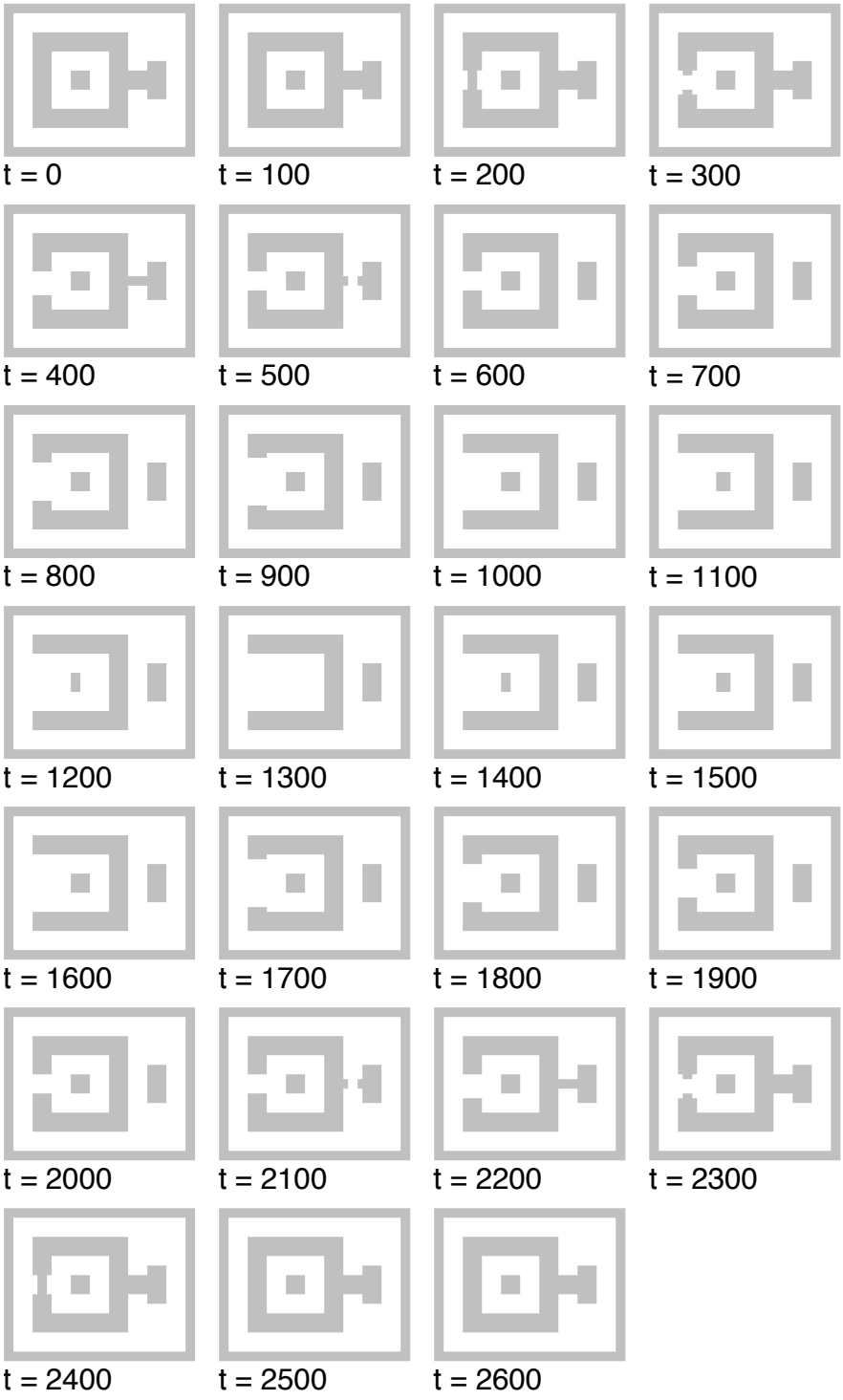



Figure 56: Complex dynamic region used in the evaluation of algorithms from chapter 5. Region dimensions are 40×32 and t indicates the time the region enters the shown configuration.

| $rid_a$ | $rid_b$ | $cid$ | $neg$ | $t_1$ | $t_2$ |
|---|---|---|---|---|---|
| -1 | 0 | -1 | FALSE | 0 | ∅ |
| 0 | 2 | a | FALSE | 100 | 500 |
| 1 | 3 | b | TRUE | 100 | 300 |
| 2 | 4 | c | FALSE | 100 | 500 |
| 0 | 7 | a | FALSE | 500 | 1000 |
| 6 | 7 | a | FALSE | 500 | 1000 |
| 0 | 6 | a | FALSE | 500 | 1000 |
| 17 | 7 | c | FALSE | 500 | 1000 |
| 0 | 7 | d | FALSE | 1000 | 1800 |
| 17 | 7 | d | FALSE | 1000 | 1300 |
| 0 | 6 | d | FALSE | 1000 | 1800 |
| 6 | 7 | d | FALSE | 1000 | 1800 |
| 0 | 4 | d | FALSE | 1000 | 1300 |
| 7 | 8 | d | FALSE | 1400 | 1800 |
| 0 | 8 | d | FALSE | 1400 | 1800 |
| 0 | 7 | e | FALSE | 1800 | 2200 |
| 0 | 6 | e | FALSE | 1800 | 2200 |
| 6 | 7 | e | FALSE | 1800 | 2200 |
| 7 | 8 | f | FALSE | 1800 | 2200 |
| 0 | 9 | e | FALSE | 2200 | ∅ |
| 8 | 9 | f | FALSE | 2200 | ∅ |
| 10 | 11 | g | TRUE | 2400 | ∅ |

Table 19: Simplified dynamic maptree table based on Figure 56 where split detected at 200, unenclose transition at 500, disappearance at 800, appearance at 900, enclose transition at 1300, and merge detected at 1500.

| $w$ | $p_1$ | $p_2$ | $split$ | $t$ |
|---|---|---|---|---|
| 5 | 1 | 3 | FALSE | 300 |
| 2 | 6 | 7 | TRUE | 500 |
| d | a | c | FALSE | 1000 |
| d | e | f | TRUE | 1800 |
| 9 | 6 | 7 | FALSE | 2200 |
| 5 | 10 | 11 | TRUE | 2400 |

Table 20: Change table based on Figure 56 where split detected at 200, unenclose transition at 500, enclose transition at 1300, and merge detected at 1500.

| *cid* | *rid* | $t_1$ | $t_2$ |
|---|---|---|---|
| a | 1 | 100 | 300 |
| b | 2 | 100 | 300 |
| c | 3 | 100 | 300 |
| a | 5 | 300 | 1000 |
| c | 5 | 300 | 1000 |
| d | 5 | 1000 | 1800 |
| e | 5 | 1800 | 2400 |
| f | 5 | 1800 | 2400 |
| e | 10 | 2400 | $\varnothing$ |
| f | 11 | 2400 | $\varnothing$ |
| g | 9 | 2400 | $\varnothing$ |

Table 21: Label table based on Figure 56 where split detected at 200, unenclose transition at 500, enclose transition at 1300, and merge detected at 1500.

[7] A. Galton, *Qualitative spatial change*. Oxford University Press, 2000.

[8] M. Duckham, *Decentralized Spatial Computing: Foundations of Geosensor Networks*. Springer Publishing Company, Incorporated, 2012.

[9] A. Both, M. Duckham, P. Laube, T. Wark, and J. Yeoman, "Decentralized Monitoring of Moving Objects in a Transportation Network Augmented with Checkpoints," *The Computer Journal*, vol. 56, no. 12, pp. 1432–1449, Dec. 2013. [Online]. Available: http://comjnl.oxfordjournals.org/content/56/12/1432

[10] A. Both, W. Kuhn, and M. Duckham, "Spatiotemporal Braitenberg Vehicles," in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. SIGSPATIAL'13. New York, NY, USA: ACM, Nov. 2013, pp. 74–83. [Online]. Available: http://doi.acm.org/10.1145/2525314.2525344

[11] M. P. Dube and M. J. Egenhofer, "Surrounds in Partitions," in *Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. SIGSPATIAL '14. New York, NY, USA: ACM, 2014, pp. 233–242. [Online]. Available: http://doi.acm.org/10.1145/2666310.2666380

[12] A. Both and M. Duckham, "Qualitative Spatial Structure in Complex Areal Objects Using Location-Free, Mobile Geosensor Networks," in *2013 IEEE 13th International Conference on Data Mining Workshops (ICDMW)*, Dec. 2013, pp. 978–985.

[13] A. G. Cohn and J. Renz, "Qualitative spatial representation and reasoning," *Foundations of Artificial Intelligence*, vol. 3, pp. 551–596, 2008.

[14] S. Nittel, N. Trigoni, K. Ferentinos, F. Neville, A. Nural, and N. Pettigrew, "A drift-tolerant model for data management in ocean sensor networks," in *International Workshop on Data Engineering for Wireless and Mobile Access*, 2007, pp. 49–58.

[15] A. G. Cohn and N. M. Gotts, "The 'egg-yolk'representation of regions with indeterminate boundaries," *Geographic objects with indeterminate boundaries*, vol. 2, pp. 171–187, 1996.

[16] D. A. Randell and A. G. Cohn, "Modelling Topological and Metrical Properties in Physical Processes." *KR*, vol. 89, pp. 357–368, 1989. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.35.9942&rep=rep1&type=pdf

[17] D. A. Randell, Z. Cui, and A. G. Cohn, "A spatial logic based on regions and connection." *KR*, vol. 92, pp. 165–176, 1992. [Online]. Available: http://www.researchgate.net/profile/Anthony_Cohn/publication/221393453_A_Spatial_Logic_based_on_Regions_and_Connection/links/0912f50cbb29aa483d000000.pdf

[18] A. G. Cohn, B. Bennett, J. Gooday, and N. M. Gotts, "Qualitative Spatial Representation and Reasoning with the Region Connection Calculus," *GeoInformatica*, vol. 1, no. 3, pp. 275–316, Oct. 1997. [Online]. Available: http://link.springer.com.ezp.lib.unimelb.edu.au/article/10.1023/A%3A1009712514511

[19] M. J. Egenhofer and R. D. Franzosa, "Point-set topological spatial relations," *International Journal of Geographical Information System*, vol. 5, no. 2, pp. 161–174, 1991.

[20] M. J. Egenhofer and J. Herring, "Categorizing binary topological relations between regions, lines, and points in geographic databases," *The*, vol. 9, pp. 94–1, 1992. [Online]. Available: http://www.spatial.cs.umn.edu/Courses/Spring10/8715/papers/MSD11_egenhofer_herring.pdf

[21] M. J. Egenhofer‡, E. Clementini, and P. d. Felice, "Topological relations between regions with holes," *International Journal of Geographical Information Systems*, vol. 8, no. 2, pp. 129–142, Mar. 1994. [Online]. Available: http://dx.doi.org/10.1080/02693799408901990

[22] M. Schneider and T. Behr, "Topological Relationships Between Complex Spatial Objects," *ACM Trans. Database Syst.*, vol. 31, no. 1, pp. 39–81, Mar. 2006. [Online]. Available: http://doi.acm.org/10.1145/1132863.1132865

[23] C. Freksa, "Temporal reasoning based on semi-intervals," *Artificial Intelligence*, vol. 54, no. 1–2, pp. 199–227, Mar. 1992. [Online]. Available: http://www.sciencedirect.com/science/article/pii/000437029290090K

[24] A. Galton, "Towards a qualitative theory of movement," in *Spatial Information Theory A Theoretical Basis for GIS*, ser. Lecture Notes in Computer Science, A. U. Frank and W. Kuhn, Eds. Springer Berlin Heidelberg, 1995, no. 988, pp. 377–396. [Online]. Available: http://link.springer.com.ezp.lib.unimelb.edu.au/chapter/10.1007/3-540-60392-1_25

[25] M. J. Egenhofer and K. K. Al-Taha, "Reasoning about gradual changes of topological relationships," in *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, ser. Lecture Notes in Computer Science, A. U. Frank, I. Campari, and U. Formentini, Eds. Springer Berlin Heidelberg, 1992, no. 639, pp. 196–219. [Online]. Available: http://link.springer.com.ezp.lib.unimelb.edu.au/chapter/10.1007/3-540-55966-3_12

[26] J. Jiang and M. Worboys, "Event-based topology for dynamic planar areal objects," *International Journal of Geographical Information Science*, vol. 23, no. 1, pp. 33–60, Jan. 2009. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/13658810802577247

[27] M. Worboys, "Modeling Indoor Space," in *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Indoor Spatial Awareness*, ser. ISA '11. New York, NY, USA: ACM, 2011, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2077357.2077358

[28] ——, "The Maptree: A Fine-Grained Formal Representation of Space," in *Geographic Information Science*, ser. Lecture Notes in Computer Science, N. Xiao, M.-P. Kwan, M. Goodchild, and S. Shekhar, Eds. Springer Berlin Heidelberg, Jan. 2012, vol. 7478, pp. 298–310. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33024-7_22

[29] M. F. Worboys and P. Bofakos, "A canonical model for a class of areal spatial objects," in *Advances in Spatial Databases*. Springer, 1993, pp. 36–52.

[30] A. Rosenfeld, "Adjacency in digital pictures," *Information and Control*, vol. 26, no. 1, pp. 24–33, Sep. 1974. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0019995874906962

[31] E. Costanza and J. Robinson, "A Region Adjacency Tree Approach to the Detection and Design of Fiducials." 2003, pp. 63–69. [Online]. Available: http://eprints.soton.ac.uk/270958/

[32] D. E. Muller and F. P. Preparata, "Finding the intersection of two convex polyhedra," *Theoretical Computer Science*, vol. 7, no. 2, pp. 217–236, 1978. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0304397578900518

[33] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf, *Computational geometry*. Springer, 2000.

[34] J. Stell and M. Worboys, "Relations between adjacency trees," *Theoretical Computer Science*, vol. 412, no. 34, pp. 4452–4468,

Aug. 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397511003276

[35] M. Worboys, "Using maptrees to characterize topological change," in *Spatial Information Theory*. Springer, 2013, pp. 74–90.

[36] M. Duckham, D. Nussbaum, J.-R. Sack, and N. Santoro, "Efficient, Decentralized Computation of the Topology of Spatial Regions," *IEEE Transactions on Computers*, vol. 60, no. 8, pp. 1100–1113, Aug. 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5551124

[37] M. Worboys and M. Duckham, "Monitoring qualitative spatiotemporal change for geosensor networks," *International Journal of Geographical Information Science*, vol. 20, no. 10, pp. 1087–1108, 2006.

[38] J. Jiang and M. Worboys, "Detecting basic topological changes in sensor networks by local aggregation," in *GIS: Proceedings of the ACM International Symposium on Advances in Geographic Information Systems*, 2008, pp. 13–22.

[39] C. Farah, C. Zhong, M. Worboys, and S. Nittel, *Detecting topological change using a wireless sensor network*, 2008, vol. 5266 LNCS.

[40] M. J. Sadeq, "In-network Detection of Topological Change of Regions with a Wireless Sensor Network," Ph.D. dissertation, University of Melbourne, Department of Geomatics, 2009.

[41] M. Shi and S. Winter, "Detecting change in snapshot sequences," in *Geographic Information Science*. Springer, 2010, pp. 219–233.

[42] J. Jiang, M. Worboys, and S. Nittel, "Qualitative change detection using sensor networks based on connectivity information," *GeoInformatica*, vol. 15, no. 2, pp. 305–328, 2011.

[43] M. Duckham, J. Stell, M. Vasardani, and M. Worboys, *Qualitative change to 3-valued regions*, 2010, vol. 6292 LNCS.

[44] M.-H. Jeong and M. Duckham, "Decentralized querying of topological relations between regions monitored by a coordinate-free geosensor network," *GeoInformatica*, vol. 17, no. 4, pp. 669–696, Feb. 2013. [Online]. Available: http://link.springer.com.ezp.lib.unimelb.edu.au/article/10.1007/s10707-012-0174-7

[45] M.-H. Jeong, M. Duckham, A. Kealy, H. J. Miller, and A. Peisker, "Decentralized and coordinate-free computation of critical points and surface networks in a discretized scalar field," *International Journal of Geographical Information Science*, vol. 28, no. 1, pp. 1–21, Jan. 2014. [Online]. Available: http://dx.doi.org/10.1080/13658816.2013.801578

[46] M. H. Jeong, "Qualitative characteristics of fields monitored by a resource-constrained geosensor network," Ph.D. dissertation, 2014. [Online]. Available: http://minerva-access.unimelb.edu.au/handle/11343/40761

[47] P. Laube, M. Kreveld, and S. Imfeld, "Finding REMO—detecting relative motion patterns in geospatial lifelines," *Developments in Spatial Data Handling*, pp. 201–215, 2005.

[48] S. Dodge, R. Weibel, and A.-K. Lautenschütz, "Towards a taxonomy of movement patterns," *Information visualization*, vol. 7, no. 3-4, pp. 240–252, 2008.

[49] G. Andrienko, N. Andrienko, U. Demsar, D. Dransch, J. Dykes, S. I. Fabrikant, M. Jern, M.-J. Kraak, H. Schumann, and C. Tominski, "Space, time and visual analytics," *International Journal of Geographical Information Science*, vol. 24, no. 10, pp. 1577–1600, 2010.

[50] J. Gudmundsson, P. Laube, and T. Wolle, "Computational movement analysis," in *Springer Handbook of Geographic Information*. Springer, 2012, pp. 423–438.

[51] J. Gudmundsson and M. van Kreveld, "Computing longest duration flocks in trajectory data," in *Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*. ACM, 2006, pp. 35–42.

[52] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle, "Reporting flock patterns," *Computational Geometry*, vol. 41, no. 3, pp. 111–125, 2008.

[53] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen, "Discovery of convoys in trajectory databases," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 1068–1080, 2008.

[54] T. Shirabe, "Correlation analysis of discrete motions," in *Geographic Information Science*. Springer, 2006, pp. 370–382.

[55] M. Andersson, J. Gudmundsson, P. Laube, and T. Wolle, "Reporting leadership patterns among trajectories," in *Proceedings of the 2007 ACM symposium on Applied computing*. ACM, 2007, pp. 3–7.

[56] ——, "Reporting leaders and followers among trajectories of moving point objects," *GeoInformatica*, vol. 12, no. 4, pp. 497–528, 2008.

[57] P. Laube, M. Duckham, and T. Wolle, "Decentralized movement pattern detection amongst mobile geosensor nodes," in *Geographic Information Science*. Springer, 2008, pp. 199–216.

[58] P. Laube, M. Duckham, and M. Palaniswami, "Deferred decentralized movement pattern mining for geosensor networks," *International Journal of Geographical Information Science*, vol. 25, no. 2, pp. 273–292, 2011.

[59] J. Yeoman and M. Duckham, "Decentralized network neighborhood information collation and distribution for convoy detection," 2012. [Online]. Available: http://www.giscience.org/past/2012/proceedings/abstracts/giscience2012_paper_172.pdf

[60] A. Savvides, C.-C. Han, and M. B. Strivastava, "Dynamic fine-grained localization in ad-hoc networks of sensors," in *Proceedings of the 7th annual international conference on Mobile computing and networking*. ACM, 2001, pp. 166–179.

[61] T. He, C. Huang, B. M. Blum, J. A. Stankovic, and T. Abdelzaher, "Range-free localization schemes for large scale sensor networks," in *Proceedings of the 9th annual international conference on Mobile computing and networking*. ACM, 2003, pp. 81–95.

[62] D. Niculescu and B. Nath, "DV based positioning in ad hoc networks," *Telecommunication Systems*, vol. 22, no. 1-4, pp. 267–280, 2003.

[63] R. Nagpal, H. Shrobe, and J. Bachrach, "Organizing a global coordinate system from local information on an ad hoc sensor network," in *Information Processing in Sensor Networks*. Springer, 2003, pp. 333–348.

[64] J. G. Lim and S. V. Rao, "Mobility-enhanced positioning in ad hoc networks," in *Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE*, vol. 3. IEEE, 2003, pp. 1832–1837.

[65] M. Grossglauser and M. Vetterli, "Locating mobile nodes with ease: learning efficient routes from encounter histories alone," *IEEE/ACM Transactions on Networking (TON)*, vol. 14, no. 3, pp. 457–469, 2006.

[66] Q. Liu, A. Pruteanu, and S. Dulman, "GDE: A Distributed Gradient-Based Algorithm for Distance Estimation in Large-Scale Networks," 2011.

[67] ——, "Gradient-Based Distance Estimation for Spatial Computers," *The Computer Journal*, p. bxt124, Nov. 2013. [Online]. Available: http://comjnl.oxfordjournals.org/content/early/2013/11/21/comjnl.bxt124

[68] S. Merkel, S. Mostaghim, and H. Schmeck, "A study of mobility in ad hoc networks and its effects on a hop count based distance estimation," in *New Technologies, Mobility and Security (NTMS), 2012 5th International Conference on*. IEEE, 2012, pp. 1–5.

[69] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-based computation of aggregate information," in *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*. IEEE, 2003, pp. 482–491.

[70] A. Pruteanu, V. Iyer, and S. Dulman, "Faildetect: Gossip-based failure estimator for large-scale dynamic networks," in *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*. IEEE, 2011, pp. 1–6.

[71] A. Pruteanu and S. Dulman, "LossEstimate: Distributed failure estimation in wireless networks," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2785–2795, Dec. 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121212002233

[72] A. Pruteanu, V. Iyer, and S. Dulman, "ChurnDetect: a gossip-based churn estimator for large-scale dynamic networks," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 289–301.

[73] L. Ramaswamy, B. Gedik, and L. Liu, "A distributed approach to node clustering in decentralized peer-to-peer networks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 9, pp. 814–829, 2005.

[74] J. Y. Yu and P. H. Chong, "A survey of clustering schemes for mobile ad hoc networks," *IEEE Communications Surveys & Tutorials*, vol. 7, no. 1, pp. 32–48, 2005.

[75] A. Pruteanu and S. Dulman, "ASH: Tackling node mobility in large-scale networks," *Computing*, vol. 94, no. 8-10, pp. 811–832, 2012.

[76] N. Bicocchi, M. Mamei, and F. Zambonelli, "Self-organizing spatial regions for sensor network infrastructures," in *Advanced Information Networking and Applications Workshops, 2007, AINAW'07. 21st International Conference on*, vol. 2. IEEE, 2007, pp. 66–71.

[77] U. Wilensky, "NetLogo," Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, http://ccl.northwestern.edu/netlogo/, 1999. [Online]. Available: http://ccl.northwestern.edu/netlogo/

[78] N. Santoro, *Design and analysis of distributed algorithms*. Wiley-Interscience, 2006, vol. 56.

[79] R. Milner, *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, May 1999.

[80] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, Apr. 1996.

[81] S. Dulman, P. Havinga, and J. Hurink, "Wave leader election protocol for wireless sensor networks," 2002.

[82] F. Bartumeus, M. G. E. da Luz, G. M. Viswanathan, and J. Catalan, "Animal Search Strategies: A Quantitative Random-Walk Analysis," *Ecology*, vol. 86, no. 11, pp. 3078–3087, Nov. 2005. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1890/04-1806/abstract

[83] G. Ramos-Fernández, J. L. Mateos, O. Miramontes, G. Cocho, H. Larralde, and B. Ayala-Orozco, "Lévy walk patterns in the foraging movements of spider monkeys (Ateles geoffroyi)," *Behavioral Ecology and Sociobiology*, vol. 55, no. 3, pp. 223–230, 2004.

[84] D. W. Sims, E. J. Southall, N. E. Humphries, G. C. Hays, C. J. Bradshaw, J. W. Pitchford, A. James, M. Z. Ahmed, A. S. Brierley, and M. A. Hindell, "Scaling laws of marine predator search behaviour," *Nature*, vol. 451, no. 7182, pp. 1098–1102, 2008.

[85] B. Sundararaman, U. Buy, and A. D. Kshemkalyani, "Clock synchronization for wireless sensor networks: a survey," *Ad Hoc Networks*, vol. 3, no. 3, pp. 281–323, May 2005. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1570870505000144